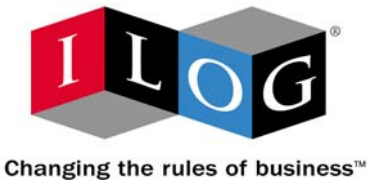


ILOG Solver 6.0

Debugger User's Manual

October 2003

Copyright © 1987-2003, ILOG, S. A. — All rights reserved.

Contents

Preface **Before You Begin 7**

About ILOG Solver Debugger 7

What You Need to Know 7

What This Manual Contains 7

Notation Used in This Manual 8

Where to Get More Information 8

Users' Mailing List 9

Web Site 9

Chapter 1 **Debugging and Performance Tuning for Solver-based Applications 11**

Installation 12

Prerequisites for Using Solver Debugger 12

The Distribution Directories 12

Examples Provided 13

Instrumenting Your Own Code 14

Customizing a Pure ILOG Solver C++ Application 14

Customizing a Pure ILOG Scheduler C++ Application 15

Setting Up ILOG Solver Debugger 17

Launching the GUI 17

Connecting an Application 18
Debugger Commands 20
Visualizing the Search 21
The Search Tree 22
Monitoring Decisions 25
Monitoring Solution Restoring 28
Monitoring Restart 28
Monitoring Local Search 28
Monitoring Nested Search 29
Visualizing Propagation 30
Events 30
Using the Propagation Spy 30
Tracing the Initial Propagation 32
Tracing the Propagation at a Specific Node. 32
Visualizing Variable Domains. 33
Visualizing Activity Domains (Scheduler-Specific) 36
Combining Search Tree Visualization and Propagation Information 38
Creating the Christmas Tree 38
Interpreting the Christmas Tree 40
Using the Constraint Profiler 40
Using the Drawing Board 41
Basic Use of the Drawing Board 42
Advanced Use of the Drawing Board 43
Improving your Application 47
Analyzing Right Subtrees. 47
Tutorial: Tuning the Propagation in the debuggolomb Example 49

Solver Debugger API 55

IlcAddOnDebugger. 57
IlcDebugger 59
IlcDrawingBoard 63
IlcSchedulerDebugger 73

IlcSolverDebugger 75
IlcVisualizeOption. 79

Index 81

Before You Begin

This manual provides you with all the information you need to use ILOG Solver Debugger.

About ILOG Solver Debugger

The ILOG Solver Debugger is composed of a GUI and a set of C++ libraries that enable you to debug and tune the performance of C++ applications written on top of ILOG Solver and ILOG Scheduler.

What You Need to Know

This manual assumes that you are familiar with the UNIX or PC environment in which you are going to use ILOG Solver Debugger, including its particular windowing system.

What This Manual Contains

Chapter 1, *Debugging and Performance Tuning for Solver-based Applications*, shows how you can use the graphical features of Solver Debugger to debug and tune the performance of C++ applications.

Notation Used in This Manual

The following typographic conventions apply throughout this manual:

- code extracts and file names are written in `this typeface`.
- entries to be made by the user are written <in angle brackets>
- commands appear as: File>Open.

Where to Get More Information

For technical support of ILOG Solver Debugger, contact your local distributor, or, if you are a direct ILOG customer, contact:

Region	E-mail	Telephone	Fax
France	solver-support@ilog.fr	0 800 09 27 91 (numéro vert) +33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Germany	solver-support@ilog.de	+49 6172 40 60 33	+49 6172 40 60 10
Spain	solver-support@ilog.es	+34 91 710 2480	+34 91 372 9976
United Kingdom	solver-support@ilog.co.uk	+44 (0)1344 661 630	+44 (0)1344 661 601
Rest of Europe	solver-support@ilog.fr	+33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Japan	solver-support@ilog.co.jp	+81 3 5211 5770	+81 3 5211 5771
Singapore	solver-support @ilog.com.sg	+65 6773 06 26	+65 6773 04 39
North America	solver-support@ilog.com	1-877-ILOG-TECH 1-877-8456-4832 (toll free) or 1-650-567-8080	+1 650 567 8001

We encourage you to use e-mail for faster, better service.

Users' Mailing List

The electronic mailing list `solver-list@ilog.fr` is available for you to share your development experience with other Solver users. This list is not moderated, but subscription is subject to an on-going maintenance contract. To subscribe to `solver-list`, send an e-mail without any subject to `solver-list-owner@ilog.fr`, with the following contents:

```
subscribe solver-list
```

your e-mail address if different from the From field

first name, last name

your location (company and country)

maintenance contract number

maintenance contract owner's last name

Web Site

On our web sites, you will find a wealth of information about constraint programming in a range of articles and conference papers explaining the theoretical background and technical features of ILOG Solver and other ILOG products.

In addition to those freely accessible pages, there are also technical support pages on our web sites. They contain FAQ (Frequently Asked/Answered Questions) and the latest patches for some of our products. Changes are posted in the product mailing list. Access to these pages is restricted to owners of an on-going maintenance contract. The maintenance contract number and the name of the person this contract is sent to in your company will be needed for access, as explained on the login page.

All three of these sites contain the same information, but access is localized, so we recommend that you connect to the site corresponding to your location, and select the "Tech Support Web" page from the home page.

Americas:	http://www.ilog.com
Asia and Pacific Nations:	http://www.ilog.com.sg
Europe, Africa, and Middle East:	http://www.ilog.fr

1

Debugging and Performance Tuning for Solver-based Applications

The ILOG Solver Debugger is composed of a GUI and a set of C++ libraries. The C++ libraries enable a C++ application written on top of ILOG Solver or ILOG Scheduler to connect to the Debugger GUI in order to monitor, trace, and debug its execution or tune its performance. ILOG Solver Debugger contains graphic facilities to monitor the search (Search Tree), the propagation (Propagation Spy and Variable Domains), and to combine search and propagation information (Christmas Tree). It provides a Constraint Profiler to obtain statistics, by constraint, about constraint calls and efficiency in terms of domain reduction. It also enables you to create your own 2D animated graphic representation of the problem being solved (Drawing Board).

This chapter explains how to connect and use ILOG Solver Debugger. It includes the following sections:

- Installation
- Instrumenting Your Own Code
- Visualizing the Search
- Visualizing Propagation
- Combining Search Tree Visualization and Propagation Information
- Using the Constraint Profiler
- Using the Drawing Board
- Improving your Application

Installation

In order to start using Solver Debugger, you need to know:

- Prerequisites for Using Solver Debugger
- The Distribution Directories
- Examples Provided

Prerequisites for Using Solver Debugger

To use the Debugger library, you need:

- ILOG Solver 6.0
- ILOG Concert 2.0

The only add-on for which specific features are supported is ILOG Scheduler 6.0.

The Distribution Directories

The ILOG Solver Debugger and its add-on the ILOG Scheduler Debugger are to be installed in the same root directory as the ILOG Solver product and the ILOG Scheduler product respectively (typically `ilog/solver60` and `ilog/sched60`).

The distribution structure includes the following directories, where `<solverdir>` represents your Solver installation directory and `<scheddir>` your Scheduler installation:

- `<solverdir>/bin/<platform>/<libformat>`

  contains the ILOG Solver Debugger GUI, `debugger.exe`
- `<solverdir>/help`

  contains the online documentation
- `<solverdir>/examples/<platform>/<libformat>/debug*`

  contains C++ examples based on ILOG Solver
- `<solverdir>/examples/src/debug*`

  contains C++ source files for the examples
- `<solverdir>/include/ilsolver`

  contains the additional include files needed for using the debugger with ILOG Solver
- `<solverdir>/lib`

  contains the libraries `ildebug.lib` and `ilsolverdebug.lib`

- `<scheddir>/examples/<platform>/<libformat>/debug*`

  contains C++ examples based on ILOG Scheduler

- `<scheddir>/examples/src/debug*`

  contains C++ source files for the examples

- `<scheddir>/include/ilsched`

  contains the additional include files needed for using the debugger with ILOG Scheduler

- `<scheddir>/lib`

  contains the additional library `ilschedulerdebug.lib`

Examples Provided

Before using the Debugger facilities on your application, you can become familiar with them by testing the examples provided as part of the Solver Debugger distribution. All the examples contain `Ilo` objects.

To compile the examples on Solaris and Linux platforms, use the command:

```
make debuggersamples
```

in the directory:

```
<product>/examples/<platform>/<libformat>
```

Solver Examples

- debugnqueen

  This is a pure ILOG Solver C++ example. This well-known problem involves placing n queens on a chessboard in such a way that none of them can capture any other using the conventional moves allowed to a queen. In other words, the problem is to select n squares on a chessboard so that any pair of selected squares is never aligned, either vertically or horizontally. This example uses a Drawing Board to represent the chessboard.

- debuggolomb

  This example is well-suited to propagation analysis with different levels of pruning for the `alldiff` constraint. It is a good example of the use of the Christmas Tree and the Propagation Spy. The goal of the debuggolomb problem is to find a set of values representing the graduations of a rule such that the difference between each pair of graduations is always distinct, and such that the length of the rule is minimal.

- debuglsknight

  This example shows how to use the Debugger with local search. It consists in placing as few knights as possible on a chessboard in such a way that each square is controlled—attacked or occupied—by at least one knight.

- debugpheasant

  This example is useful for tracing the Initial Propagation. This problem consists in finding the number of pheasants and rabbits in a field, knowing that there are 20 heads and 56 legs overall.

Scheduler Example

- debugjobshopm

  The aim of this problem is to schedule a number of jobs on a set of machines in order to minimize completion time, called `makespan`. Each job is a sequence of tasks and each task represents a machine. This example enables you to see the Activity Domains window, the Gantt chart and specific scheduling events in the Propagation Spy. In the code of debugjobshopm you will find examples of conditional breakpoints, Debugger disabling, and the automatic result display switched off.

Instrumenting Your Own Code

This section explains how to customize the Debugger for ILOG Solver or ILOG Scheduler applications.

- Customizing a Pure ILOG Solver C++ Application
- Customizing a Pure ILOG Scheduler C++ Application

Customizing a Pure ILOG Solver C++ Application

To customize the Debugger for a pure ILOG Solver C++ application, proceed as follows. You need one include file before the solver declaration:

```
#include <ilsolver/solverdebugger.h>

IloEnv env;
try {
  IloSolver solver(env);
```

1. Instantiate the Debugger and let it connect your application to the GUI before model extraction.

   ```
   IlcSolverDebugger debugger(solver);
   ```

2. State the model using `Ilo` modeling objects.

   ```
   IloModel model(env);
   ```

3. Name your `Ilo` objects (`setName` API).

   ```
   myVar.setName("myVar");
   ```

4. Register the variables you are interested in for model browsing, domain monitoring, etc. (Optional).

```
debugger.registerVariable(myVar);
```

   If you want domain visualization, specify it.

```
debugger.registerVariable(myVar2, IloTrue, IloTrue, IlcVisualizeDomain);
```

5. Extract the model `Ilo` objects as `Ilc` objects.

```
solver.extract(model);
```

6. Initialize the Debugger for each running session and solve by placing your optimization loop inside the debugging loop.

```
  while (debugger.initialize()) { // debugging loop
    solver.startNewSearch(myGoal);
    while (solver.next()) { // optimization loop
      ostrstream text;
      text << solver.getValue(makespan)<< endl << ends;
      debugger.sendSolution(text.str());
    }
    solver.endSearch();
  }
```

7. Close the connection to the GUI and release memory.

```
    debugger.close();
  } catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
```

Customizing a Pure ILOG Scheduler C++ Application

To customize the Debugger for a pure ILOG Scheduler C++ application, proceed as follows. You need two include files before the solver declaration:

```
#include <ilsolver/solverdebugger.h>
#include <ilsched/schedulerdebugger.h>

try {
  IloEnv env;
  IloSolver solver(env);
```

1. Instantiate the Debugger.

```
    IlcSolverDebugger debugger(solver);
```

2. State the model using `Ilo` modeling objects.

```
IloModel model(env);
IloSchedulerEnv schedEnv(env);
IloInt horizon = 0;
IloInt k;
for (k = 0; k < numberOfActivities; k++)
  horizon += durations[k];
```

> ***Warning:*** *Make sure that you set the global horizon for every activity:*
>
> `schedEnv.setHorizon(horizon);`

3. Name your `Ilo` objects (`setName` API).

4. Create a Scheduler Debugger to handle specific panels.

```
IlcScheduler schedule(solver);
IlcSchedulerDebugger schedDebug(debugger,schedule);
```

5. Register the `Ilo` objects with the Solver Debugger and the Scheduler Debugger.

```
debugger.registerVariable(makespan,IloTrue,IloTrue,IlcVisualizeDomain);
schedDebug.registerActivity(activity,IloTrue,IloTrue,IlcVisualizeDomain);
```

6. Inform the Activity Panel of the horizon to display.

```
schedDebugger.setHorizon(horizon);
```

7. Extract the model `Ilo` objects as `Ilc` objects.

```
solver.extract(model);
```

8. Initialize the Debugger for each running session and solve by placing your optimization loop inside the debugging loop.

```
while (debugger.initialize()) { // debugging loop
  solver.startNewSearch(myGoal);
  while (solver.next()) { // optimization loop
   ostrstream text;
   text << solver.getValue(makespan)<< endl << ends;
   debugger.sendSolution(text.str());
  }
  solver.endSearch();
}
env.end();
```

9. Close the connection to the GUI and release memory.

```
  debugger.close();
  } catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
}
```

Setting Up ILOG Solver Debugger

In this section you will see how to launch the Debugger GUI and connect an application to it.

- Launching the GUI
- Connecting an Application

Launching the GUI

In a command prompt window, go to the bin subdirectory where ILOG Solver is installed and type:

```
debugger
```

On Windows you can use the shortcut provided in:

```
Start Menu>ILOG>ILOG Solver>ILOG Solver Debugger
```

By default, the port used for communication between the GUI and your application is 4011.

If you want to use a specific port number, pass, for example, `-listensocket 8000` as an argument.

The GUI starts in initializing mode with a red patch in the bottom right corner of the window. ILOG Solver Debugger is waiting for your application to connect.

Connecting an Application

To connect an example of an application, go to the directory:

`<solverdir>/examples/`

Compile and link a debugger example (such as debugnqueen). To start the application:

- either select File>Launch application
- or run it from the command line.

The GUI switches to idle mode with a blue patch in the bottom right corner of the window. The Run button is now active. The GUI is ready for you to debug the application.

> ***Note:*** *When using the debuggolomb example, pass it two integer parameters: the size of the problem and the filtering level of the* `alldiff` *constraint.*
> `debuggolomb 6 0` *(0 = basic filtering level)*
> `debuggolomb 6 1` *(1 = medium filtering level)*
> `debuggolomb 6 2` *(2 = extended filtering level)*
> *You can select the arguments before starting the application with File>Arguments.*

Connecting Two Applications at the Same Time

To load two applications at the same time, proceed as follows:

1. Launch two Debugger GUIs with different socket port numbers, for example:

   ```
   debugger -listensocket 5000 &
   debugger -listensocket 5001 &
   ```

2. Connect each application to a different port number giving extra arguments to the constructor `IlcSolverDebugger`.

   In one application, type:

   ```
   IlcSolverDebugger debugger(solver, IloTrue, IloFalse, 1.0, 5000);
   ```

   In the other application, type:

   ```
   IlcSolverDebugger debugger(solver, IloTrue, IloFalse, 1.0, 5001);
   ```

Connecting an Application Remotely

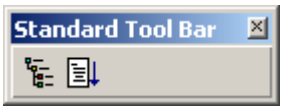
To load an application remotely, give the machine name and the port number expected by the GUI to the constructor `IlcSolverDebugger`.

```
IlcSolverDebugger debugger(solver, IloTrue, IloFalse, 1.0, 4011,
                           "NameOfMachineOnWhichTheGUIisRunning");
```

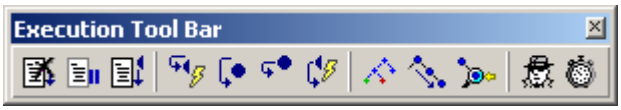
> ***Note:*** *To activate or deactivate the Debugger without recompiling, pass* `IloTrue` *or* `IloFalse` *as a second argument to the* `IlcSolverDebugger` *constructor based on a command line argument of your application.*

Debugger Commands

Once you load an application, the Run button becomes active in the Standard Tool Bar. Your application has built the optimization model and is waiting for you to trigger the solving by pressing Run. The run session is blocked in the `IlcSolverDebugger::initialize()` method.



After clicking on the Run button, the Execution Tool Bar appears. Your application returns from the `initialize()` method and solving starts.



These are the buttons that you need to control the Debugger.

- **Abort**

Stops the current computation.

- **Break**

Interrupts the execution.

- **Continue**

Continues the execution and stops tracing the propagation.

- **Step Into**

Steps into a node and traces the propagation within the node.

- **Step Over**

Steps over a node, that is, skips the propagation information within that node, and stops at the beginning of the next node visit.

- **Step Out**

Steps out of a node and avoids going through all the details of the propagation.

- **Continue Trace Till Next Node**

Traces the propagation events occurring at the current node, and stops at the beginning of the next node visit.

- **View Search Tree**

Displays a graphic representation of the search, allowing you to visualize each step of the execution.

- **Decision Stack**

Inspects the list of decisions taken, starting from the root node to the selected node.

- **Inspect Current Node**

Inspects the current choice point in the execution.

- **Propagation Spy**

Displays the trace of propagation events and the result of the propagation.

- **View Constraint Profiler**

Computes profiling statistics and displays the Constraint Profiler for the current state of computation.

Visualizing the Search

ILOG Solver Debugger provides you with a graphic representation of the search, called the Search Tree. The Search Tree allows you to visualize each step of the execution of your application. This section covers the following points:

- The Search Tree
- Monitoring Decisions
- Monitoring Solution Restoring
- Monitoring Restart
- Monitoring Local Search
- Monitoring Nested Search

The Search Tree

When you first launch an execution, a drawing of the tree appears node by node in the Search Tree viewer. If necessary, interrupt the execution with the Break button. The Debugger will stop at the beginning of a node visit.

If you previously closed the search tree views you can redisplay them by selecting the menu item View>View Search Tree, or by clicking on the corresponding button in the tool bar.

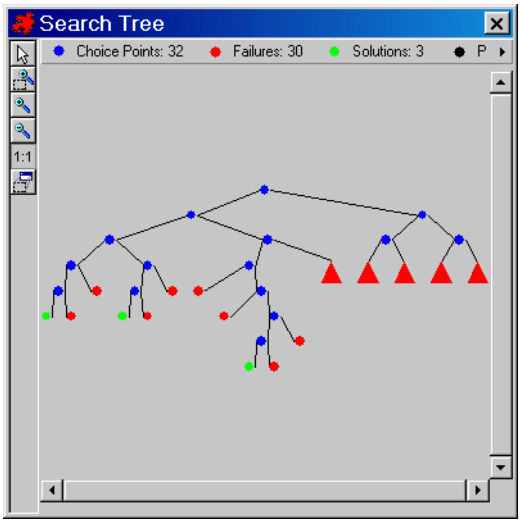
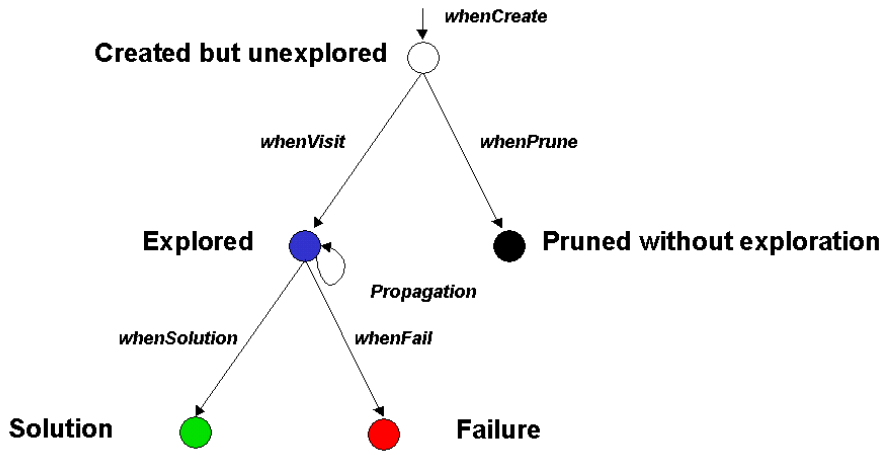


Figure 1.1 *The Search Tree for the debuggolomb problem with 5 collapsed subtrees*

The Node Life Cycle

There are five node colors that correspond to the status of a node in the Search Tree, as shown in the graphic below:



While the yellow arrow points to the root node, the algorithm is performing the initial domain reduction and the Initial Goal successively.

Interacting with a Node

Right-click on a node to:

- add or remove breakpoints
- expand or collapse a subtree
- open the node inspector

  The node inspector contains a trace of the decisions. The node inspector also contains statistics on the number of propagation events that took place and the domain reduction performed at the node currently being inspected.

Search Tree Mode Selectors

Use Search Tree mode selectors to choose the view you want for the Search Tree.

- **Selection mode**

Click on the arrow button, then double-click on a node belonging to the current branch.

- **Elastic zoom mode**

Click on this button then, with the left mouse button, trace a rectangle around the element(s) you want to enlarge. When you release the mouse button, the enlarged image fills the window.

- **Zoom in**

Click on the button with a + sign, then click in the Search Tree window. The image is increased by a factor of 2 each time you click.

- **Zoom out**

Click on the button with a – sign, then click in the Search Tree main window. The image is decreased by a factor of 2 each time you click.

- **View whole content**

Click on this button then click in the Search Tree window. The whole tree becomes visible.

- **Create new window**

Click on this button, then trace a rectangle around the element(s) you want to view in another window. A new dockable window is created, with a zoom level defined by the rectangle.

Monitoring Decisions

You can use several visualization tools to monitor the decisions within your application: the Node Inspector, the Propagation Spy, the Decision Stack, or a tooltip on the Search Tree.

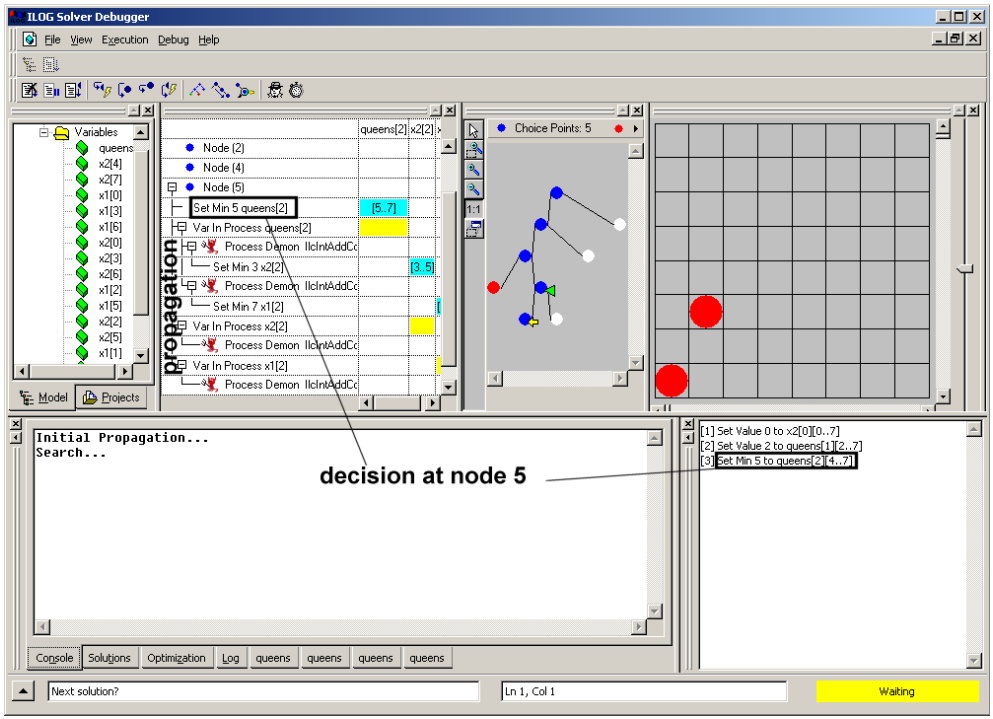


Figure 1.2 *Decision and propagation at a search tree node for the debugnqueen problem*

The interface shown above contains the elements used to monitor the decisions taken at each node in the search tree:

Current Node Inspector

Search Tree

Propagation Spy

Current Node Inspector

To use the Current Node Inspector docked panel, proceed as follows:

1. Select Stepping in Model from the Debug menu to stop at the beginning of the execution or click on Break to interrupt the execution.
2. Click on the Inspect Current Node button in the debugging tool bar.
3. The Inspect Current Node appears as a docked panel. You can monitor the decisions taken online.

Node Inspector Dialog Box

Right-click on a node to display the Node Inspector dialog box. In this dialog box, information on decisions taken at a node is only available when the exploration of the next node begins.

Propagation Spy

The Propagation Spy does not only contain the propagation events, it also contains the decision events.

> ***Note:*** *In the current implementation, the monitored decisions cannot be distinguished from pushed constraints and optimization events.*

For information on how to use the Propagation Spy, see *Using the Propagation Spy*.

Decision Stack

The Decision Stack is a panel that lists the decisions made in the current branch, starting from the root node.

When you double-click on a node, a green arrow points to the node you selected, and the Decision Stack (docked) panel displays the decisions taken in the branch starting from the root to the selected node. The description of the decision taken at a node is only available when the exploration of the next node begins.

The Decision Stack indicates the depth of the node starting from the root, displayed between brackets. For instance, in the debugnqueen example, `[48] Set Value 47 to vars[94]` corresponds to a decision taken at the node pointed to by the green arrow. `[48]` indicates the depth from the root.

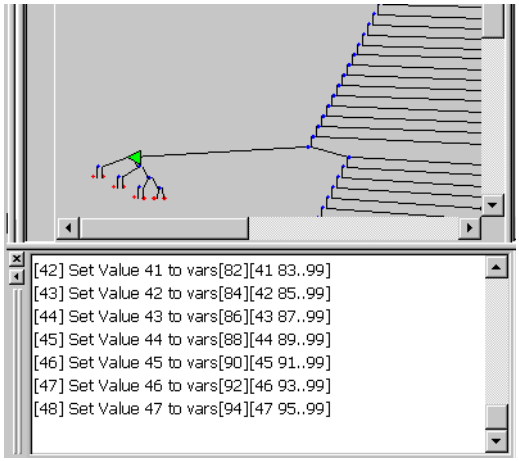

[42] Set Value 41 to vars[82][41 83..99]
[43] Set Value 42 to vars[84][42 85..99]
[44] Set Value 43 to vars[86][43 87..99]
[45] Set Value 44 to vars[88][44 89..99]
[46] Set Value 45 to vars[90][45 91..99]
[47] Set Value 46 to vars[92][46 93..99]
[48] Set Value 47 to vars[94][47 95..99]

Figure 1.3 *The Decision Stack panel for the debugnqueen problem*

> ***Note:*** *Sometimes the last line is the only visible one. Even if the vertical scroll bar appears as full, previous lines do exist. To force scrolling up, drag the cursor up inside the text.*

Tooltip on the Search Tree

To access quickly the decisions taken at a node, place the cursor over the node graphic representation or over the link above. A tooltip will appear with the decision trapped by the debugger at this node.

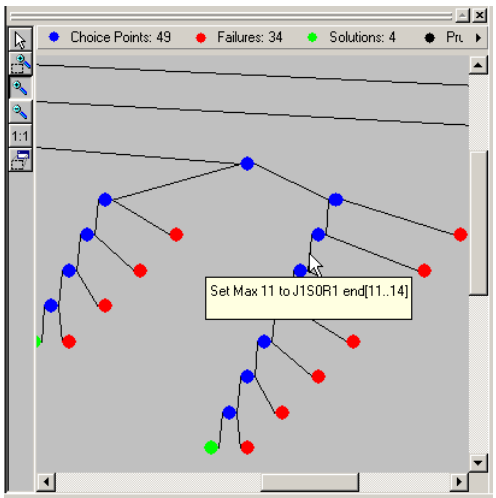


Figure 1.4 *Displaying a decision via the tooltip*

Monitoring Solution Restoring

To obtain a new Search Tree view when restoring a solution, type:

```
debugger.newSearchTreeView();
solver.solve(IloRestoreSolution(env, best));
```

To prevent the Debugger from displaying the Gantt chart automatically at each intermediate solution, type: `debugger.setAutomaticDisplay(IloFalse);`

Then call `debugger.displayresult()` when the optimal solution is restored.

Monitoring Restart

To obtain a new Search Tree view at restart, type:

```
while(debugger.initialize()) {
      solver.startNewSearch(myGoal);
      if (solver.next()) {
         ostrstream text;
         text << solver.getValue(makespan)<< endl << ends;
         debugger.sendSolution(text.str());
         debugger.newSearchTreeView();
         solver.restartSearch();
         if (solver.next()) {
            ostrstream text;
            text << solver.getValue(makespan)<< endl << ends;
            debugger.sendSolution(text.str());
        }
      }
      solver.endSearch();
}
```

You can also choose to reuse the same Search Tree view. In that case, the root node will have four branches.

Monitoring Local Search

To obtain a new Search Tree for each move, proceed as follows:

1. Type:

   ```
   IlcSolverDebugger debugger(solver);
   solver.extract(model);
   while (debugger.initialize()) {
   ```

2. Create the initial solution:

   ```
   solver.solve(IloGenerate(env, knightArr)
                && IloStoreSolution(env, sol));
   do {
   ```

3. While there are iterations, create a new Search Tree view and start a new search for each iteration:

```
while (--iter >= 0
       && debugger.newSearchTreeView() && solver.solve(move)) {
  moves++;
```

4. Store the new best solution if there is one, and display it:

```
    if (sol.getObjectiveValue() < best.getObjectiveValue()) {
      ostrstream txt;
      txt << "Move " << moves
      << ", Knights = " << sol.getObjectiveValue() << endl<< ends;
      debugger.sendSolution(txt.str());
      best.copy(sol);
    }
  }
```

5. If iterations are still left, ask `tabu` if it is complete:

```
 } while (iter > 0 && !tabu.complete());
```

6. Restore the best solution found:

```
    debugger.sendConsole("Restoring solution...\n");
    debugger.newSearchTreeView();
    solver.solve(IloRestoreSolution(env, best));
}
```

See the debuglsknight example for details on local search.

Monitoring Nested Search

To monitor nested search, type the following:

```
IlcSolverDebugger debugger(solver,IloTrue,IloFalse,1.0,4005,0,IloTrue);
ILCGOAL2(MyIlcGoal,IlcIntVarArray, a, IlcSolverDebugger, debugger) {
   debugger.beginNestedSearchTreeView();
   a.getManager().solve(IlcGenerate(a));
   debugger.endNestedSearchTreeView();
   return 0;
}
ILOCPGOALWRAPPER2(MyIloGoal, solver, IloIntVarArray, a,
                  IlcSolverDebugger, debugger) {
   return MyIlcGoal(solver.getManager(),solver.getIntVarArray(a),debugger);
}
```

Visualizing Propagation

ILOG Solver Debugger provides you with three tools that allow you to display the propagation in the GUI:

- the Propagation Spy to trace events, described in *Using the Propagation Spy*
- the Variable Domains notebook pages in the Output window, described in *Visualizing Variable Domains*
- the Activity Domains window for Scheduler applications, described in *Visualizing Activity Domains (Scheduler-Specific)*

Events

Events can take place:

- during the Initial Propagation
- during the Initial Goal before the first OR
- inside a Search Tree node

Events can be triggered as a direct consequence of:

- a constraint propagation triggered by a demon becoming active
- a constraint propagation coming from a pushed constraint
- a decision taken during the search
- an optimization event coming from the minimization or maximization of an objective function

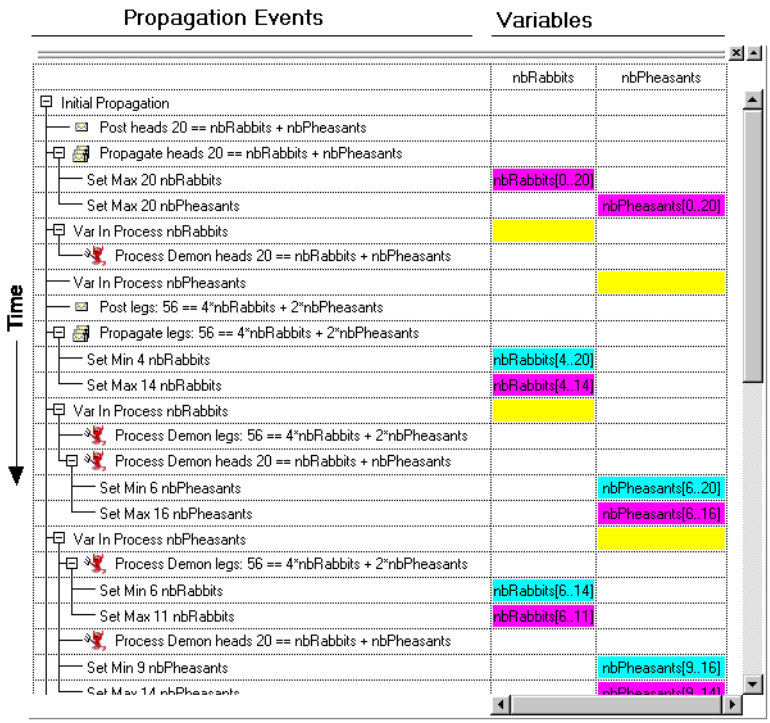
Using the Propagation Spy

The Propagation Spy is a means of displaying the trace of events and the result of the propagation in terms of domain reduction. The Propagation Spy provides a graphical representation of events trapped by the `IlcTrace` and `IlcSchedulerTrace` mechanisms.

See ILOG Solver and ILOG Scheduler documentation for details on these two classes.

The Propagation Spy is a special hierarchical sheet containing a tree hierarchy in the first column. Each column contains the impact on one variable. Time is represented vertically, from top to bottom. The textual description of the events is displayed in the first column. When double-clicking on an event line, the columns are rearranged so that a variable of interest is displayed first.

The figure below shows what the tracing of the Initial Propagation for the Pheasants and Rabbits problem (debugpheasant) looks like in the Propagation Spy.



The cell at the junction of an event line and a variable column is colored according to the type of event and shows the result of the event in a textual form. You can follow the history of the domain reduction in that cell.

The correspondences between cell colors and the event types are:

- Red = `fail`
- Yellow = `inProcess`
- Blue = `setValue`
- Wheat = `removeValue`
- Cyan = `setMin`
- Magenta = `setMax`

Tracing the Initial Propagation

To trace the Initial Propagation of your application, proceed as follows:

1. Select “Stop At Initial Propagation” from the Debug menu. With this option, the execution will be interrupted at the beginning of the Initial Propagation.
2. Click on the Run button.
3. The Initial Propagation is now displayed in the Propagation Spy window. The following options are available:
   - To enter the propagation detail, click on the Step In button.

     - Trace all the Initial Propagation events without stopping at each event by clicking on the Continue Trace Till Next Node button.

     - Trace the Initial Propagation and stop at each event by clicking on the Step In button.

     - Leave the propagation detail at that point by clicking on the Step Out button.
   - To skip the Initial Propagation, click on the Step Over button.
4. The Debugger stops at the first search node (the first child of the root). The following options are available:
   - To visualize the decisions and the propagation events triggered within the first search node in the Propagation Spy, click on the Step In button.
   - To go to the beginning of the next node, click on the Step Over button.
   - To continue and interrupt the execution when you decide to, click on the Continue Run button and then on the Break button.
   - To abort the execution, click on the Abort button.

Tracing the Propagation at a Specific Node

Because of the nondeterministic nature of the search, it is not possible to draw the tree in advance. A second run is necessary to focus on a segment of the tree.

You can access statistical information about the Initial Propagation through the Search Tree root.

Tracing the Propagation at a Specific Node in the Search Tree

To trace the propagation at a specific node, proceed as follows:

1. Run the Debugger a first time. Skip the Initial Propagation. Wait until the end of the solving or Abort if the solving process takes a long time.

2. Add a breakpoint on a specific node by right-clicking on the node (you can also include conditional breakpoints in your code).
3. Run the execution a second time. The tree nodes become white and are progressively re-colored.
4. Step into the node to display the propagation events.

Adding Breakpoints in Your Code

You can register conditional breakpoints in your code so that as soon as a condition becomes true, the execution is interrupted. You can use three types of conditional breakpoints:

- Interrupt the execution as soon as the minimum of a variable domain is greater than or equal to a value.
- Interrupt the execution as soon as the maximum of a variable domain is lower than or equal to a value.
- Interrupt the execution as soon as a variable is bound to a specific value.

For example, in the debugjobshopm example, uncomment the following line:

```
debugger.breakIfDomainMaxLe(_makespan),60);
```

The execution will be interrupted when the maximum of the `makespan` domain is lower than or equal to the value `60`.

Conditional breakpoints are extracted by the `initialize` member function. So make sure you set the conditional breakpoints before the call to the member function `IlcSolverDebugger::initialize()`.

You can also call the member function `IlcSolverDebugger::breakExecution()` in your own goals.

Visualizing Variable Domains

You can set visualization options to follow the domains of the variables during the execution in the Output window located at the bottom of the Solver Debugger GUI.

Setting Visualization Options

To follow the domains of the variables, you can put graphical demons on your variables by passing a fourth argument to the member function `registerVariable()`.

```
debugger.registerVariable(ilovars,IloTrue,IloTrue,
                                IlcVisualizeDomain | IlcVisualizeInterval |
                                IlcVisualizeValue | IlcVisualizeBoolean,
                          "queens");
```

- To visualize the domain of an integer variable or an array of integer variables, use `IlcVisualizeDomain`.
- To visualize the lower and upper bounds of a variable or an array of variables, use `IlcVisualizeInterval`.
- To visualize the value of a variable or an array of variables, use `IlcVisualizeValue`.
- To display colored squares for Boolean variables (integer variables containing 0 in their domain), use `IlcVisualizeBoolean`.
- To disable visualization, use `IlcVisualizeDisabled`. Visualization is disabled by default.

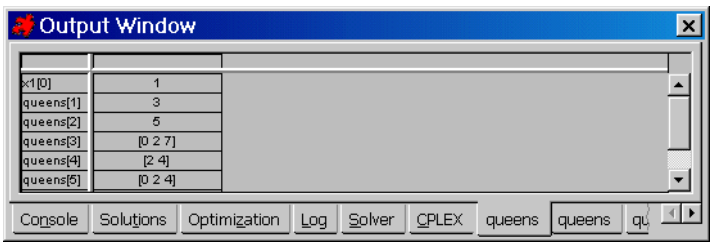
These displays are available as notebook pages in the Output window of the Solver Debugger GUI.

> ***Note:*** *You may miss some changes if a failure occurs at a node before the graphical demon is executed.*

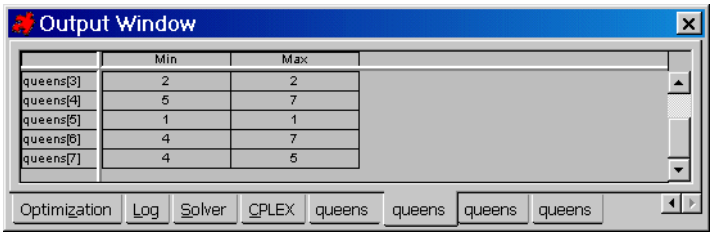
Visualizing Variable Domains in the Output Window

The following variable visualization examples are taken from the debugnqueen problem.

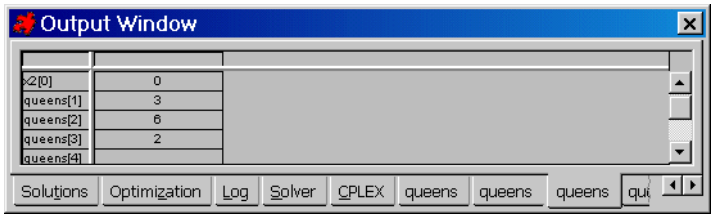
- To visualize the domains of an array of integer variables, click on the corresponding tab in the Output window. In this example, because of the presolve that occurs when a model is extracted, the `x1[0]` variable replaces the expected `queens[0]`.



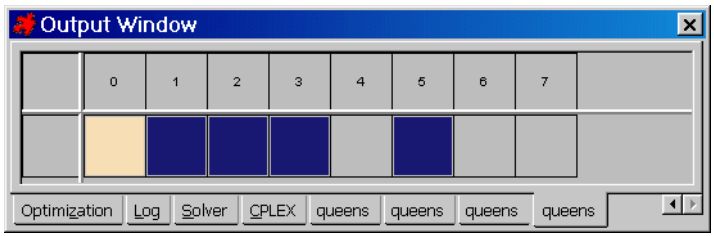
- To visualize the bounds of an array of integer variables, click on the corresponding tab in the Output window.



- To visualize the values of an array of integer variables, click on the corresponding tab in the Output window.



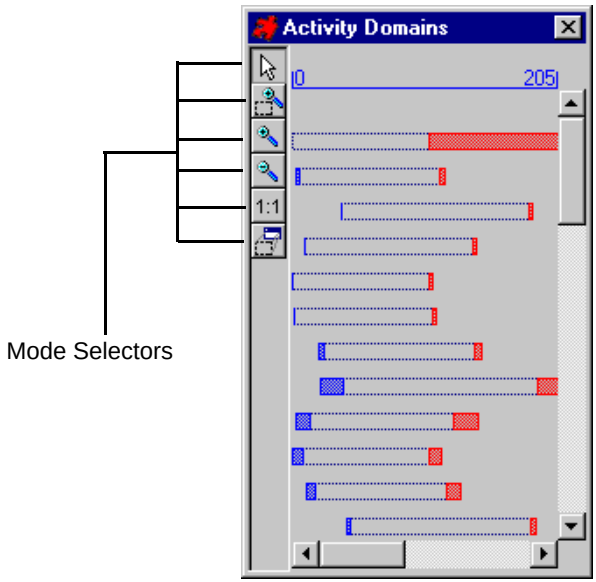
- To visualize the state of an array of integer variables interpreted as Boolean, click on the corresponding tab in the Output window. In this example, the colors have the following meaning: wheat stands for false, blue stands for true, and grey stands for undetermined.



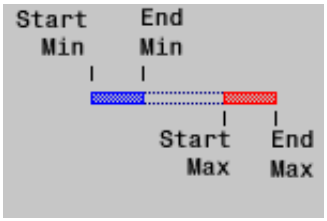
Visualizing Activity Domains (Scheduler-Specific)

Activity Domains are specific to the product ILOG Scheduler.

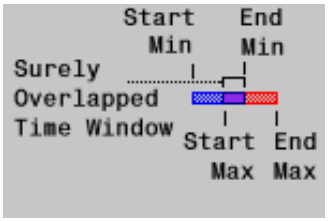
The Activity Domains dockable window shows the algorithm reducing the domains of the activities during the propagation.



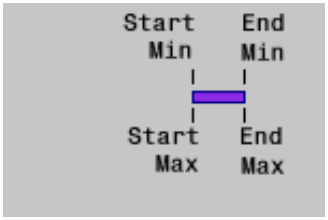
The following figures represent the evolution of Activity Domains throughout the propagation.



Because of the breakable nature of some of the activities, the `Start Min- End Min` blue rectangle and the `Start Max-End Max` red rectangle may not always be the same size.



As soon as `end min` passes `start max`, the time window between these two dates is the surely overlapped time window.



The activity is totally instantiated. When `start min` = `start max` and `end min` = `end max`, the activity schedule is totally determined.

The order in which the activities appear in the Activity Domains window is the same as the order of the calls made to the member function `IlcSchedulerDebugger::registerActivity`.

Combining Search Tree Visualization and Propagation Information

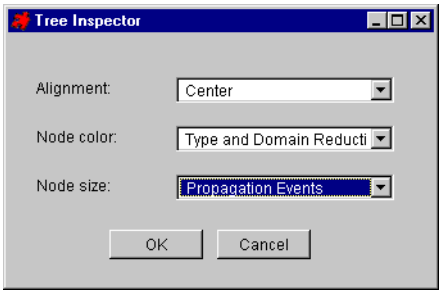
The representation of a tree combining search information and statistics on propagation events looks like a tree decorated with big and small balls. For this reason, it is called the Christmas Tree. This section deals with

- Creating the Christmas Tree
- Interpreting the Christmas Tree

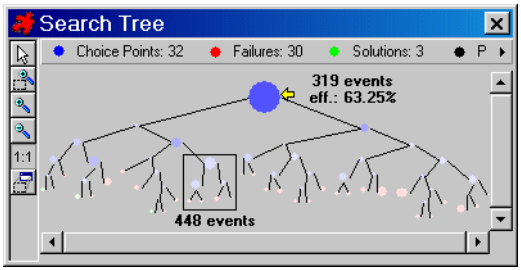
Creating the Christmas Tree

You can change the representation of the Search Tree to visualize propagation statistics by changing the color and size of the nodes. To do so:

1. Right-click in the search tree background on a node. A context-sensitive menu appears. Select the “Tree Inspector...” menu item.
2. Select Type and Domain Reduction for Node color and Propagation Events for Node size.



3. The Christmas Tree appears in the Search Tree view.



To obtain statistics on domain reduction, you must register the variables you are interested in with `IloSolverDebugger::registerVariable` for pure Solver variables and

`IlcSchedulerDebugger::registerActivity` for Scheduler activities, and pass `IloTrue` as a third argument.

Interpreting the Christmas Tree

The information added to the Search Tree includes the number of propagation events and the effective global domain reduction at each node:

- The size of the nodes becomes proportional to the number of propagation events fired at each node (which is highly correlated with the time spent at each node).
- The meaning of the colors remains unchanged, except that the color is lighter or darker depending on the effective domain reduction obtained during the propagation at this node.

The statistical information about the Initial Propagation is concentrated in the tree root. All nodes are not equal. You can detect if, in failure nodes (red or pink nodes), the failure is discovered early (small node) or late (large node).

It is important to distinguish between the number of propagation events and the global domain reduction because many propagation events can be triggered with little impact on the domain reduction.

Using the Constraint Profiler

You can access the current state of statistics at any moment during the search or the initial propagation by selecting the menu item View>Constraint Profiler, or by clicking on the Constraint Profiler button in the tool bar.

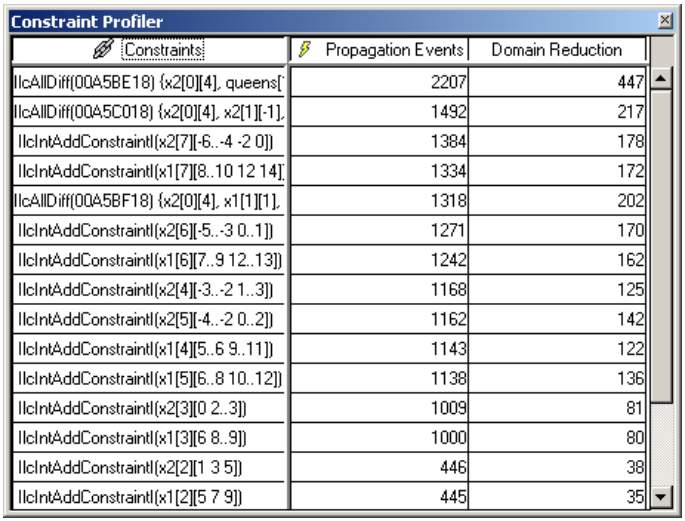


Figure 1.5 *The Constraint Profiler for the debugnqueen problem*

The Constraint Profiler displays statistics, by constraint, about the number of propagation events triggered and the effective domain reduction achieved when the corresponding constraint is active. You can double-click on a column title to sort the lines by:

- alphabetical order of constraint description
- number of propagation events triggered (default order)
- effective domain reduction achieved

A constraint can be called several times with no efficiency in terms of domain reduction. The column domain reduction corresponds to the sum of the values removed by the active constraint from integer variables. The domain reduction is evaluated by taking the difference of the value returned by `getSize()` on the variables before and after the event.

> ***Note:*** *You can copy the content of the Constraint Profiler by selecting cells while dragging the mouse and then pressing Ctrl-C. You can paste the result in Microsoft Excel or Star Office.*

Using the Drawing Board

ILOG Solver Debugger provides you with a tool that complements the predfined visualization tools: the Drawing Board. The Drawing Board allows you to define your own backtrackable 2D graphical representation to animate the search algorithm or represent solutions. The Drawing Board is a dockable window that you can float and resize.

To create a Drawing Board, instantiate an `IlcDrawingBoard` object and pass the `IlcSolverDebugger` instance as an argument.

```
IlcDrawingBoard drawingBoard(debugger);
```

You can create several instances of the class `IlcDrawingBoard`, which will result in the creation of several Drawing Board docked windows in the GUI.

The Drawing Board contains two sets of member functions:

- a set that allows you to use the Drawing Board with graphic objects that are automatically backtrackable
- a set that allows you to draw 2D graphic objects yourself in the Drawing Board without any automatic support for backtracking.

> ***Note:*** *If you previously closed a drawing board panel, you can redisplay it by selecting the menu item View>View Drawing Board.*

Basic Use of the Drawing Board

The first set of member functions enables you to connect your `Ilc` variables to parameters of the graphic object.

When all the parameters needed by a graphic object are known, that is, when the variables are bound, the graphic object is drawn. Upon backtracking, the graphic object is cleared automatically.

The member functions begin with `makeXXXX` and take `Ilc` objects as arguments (`makeRectangle`, `makeFilledPolygon`, or `makeLabel` for instance). Because this set of member functions requires `Ilc` and not `Ilo` objects, call them in a goal, for instance in the initial goal.

In the following example, an `IlcGoal` (`IlcChessBoard`) is wrapped in an `IloGoal` (`IloChessBoard`) and an`IloAndGoal` is performed on `IloChessBoard` and the ordinary `IloGenerate` goal.

Example: Drawing the Chessboard for the debugnqueen Problem

Let's call `IloChessBoard` a user-defined goal whose role is to initialize a grid representing a chessboard and ellipses indicating the locations of the queens. Here is an example of an `IlcGoal` wrapped as `IloGoal`:

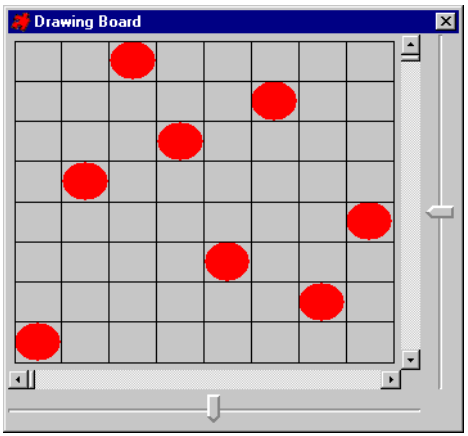
```
ILCGOAL3(IlcChessBoard, IlcInt, nqueen, IlcIntVarArray,
         queens,IlcDrawingBoard, drawingBoard) {
   drawingBoard.clean();
   drawingBoard.makeGrid(
               drawingBoard.constantInt(0),
               drawingBoard.constantInt(0),
               drawingBoard.constantInt(nqueen*10),
               drawingBoard.constantInt(nqueen*10),
               drawingBoard.constantInt(nqueen),
               drawingBoard.constantInt(nqueen),
               drawingBoard.constantInt(1));
   for(int i=0; i < nqueen; ++i) {
      IlcIntExp exp = queens[i]*10;
      drawingBoard.makeFilledEllipse(drawingBoard.constantInt(i*10), exp,
                                     drawingBoard.constantInt(10),
                                     drawingBoard.constantInt(10), "red",
                              drawingBoard.constantInt(1));
   }
   return 0;
}

ILOCPGOALWRAPPER3(IloChessBoard, solver, IloInt, nqueen, IloIntVarArray,
                  queens, IlcDrawingBoard, drawingBoard){

   return IlcChessBoard(solver, nqueen,
                        solver.getIntVarArray(queens),drawingBoard);

}
```

Here is how to start the search:

```
solver.startNewSearch(IloAndGoal(solver.getEnv(),
                                 IloChessBoard(solver.getEnv(),
                                               nqueen,ilovars,drawingBoard),
                                 IloGenerate(solver.getEnv(),ilovars)));
```

> ***Note:*** *The expected colors use X names. For example:*
> `"red","green","blue","yellow","pink","brown","magenta","cyan",`
> `"white","black","turquoise1","SeaGreen1","gold1","IndianRed1",`
> `"Sienna1","tan1","salmon1","orange1","tomato1","HotPink1","orchid2"`

The Drawing Board now displays the representation of a chessboard, with red ellipses indicating the position of the queens.



Advanced Use of the Drawing Board

The second set of member functions allows you to create and manage the graphic objects yourself.

- First create a graphic object with the member function `makeNewGraphicObject` on the class `IlcDrawingBoard`. This member function provides you with an identifier that will enable you to later refer to the graphic object.

  ```
  IloInt id = drawingBoard.makeNewGraphicObject();
  ```

- Then call the `drawXXXX` member functions. These member functions take constants as arguments. If you want to erase your graphic object, you can do it with the member function `eraseGraphic()`.

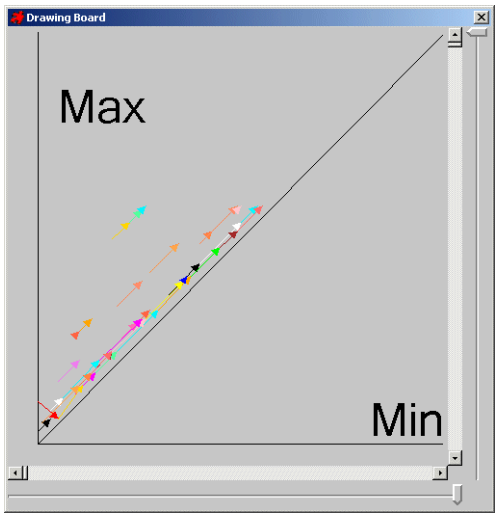
Example: Creating a New Algorithm Animation for a Scheduling Application

In the following snippet of code, the code of the debugjobshopm example is changed in order to represent the minimum and maximum values of the variables in a 2D representation.

The x-axis indicates the value of the minimum of the domain. The y-axis indicates the value of the maximum of the domain. Thus, the range of the variable domain can be represented by a 2D arrow.

When the arrow is on the diagonal, `min` equals `max`, which means that the variable is bound. A colored arrow is created between the start and end variables of a set of activities. Backtracking is handled by using reversible actions. When searching for a solution, the arrows start moving towards the diagonal. Sometimes an arrow takes a negative slope, which indicates that the domain of the start variable is no longer compatible with the domain of the end variable of the corresponding activity. A failure then occurs and the algorithm backtracks.

The following figure represents a user-defined animation of the search for a Scheduler application. The red arrow at the bottom left takes a negative slope. This arrow represents the activity responsible for a failure: its end `max` is strictly lower than its start `max`.



The following snippet of code is an extract from the debugjobshopm example.

```
const char* colors[] = {
   "red","green","blue","yellow","pink","brown","magenta","cyan","white",
   "black","turquoise1","SeaGreen1","gold1","IndianRed1","Sienna1",
   "tan1","salmon1","orange1","tomato1","HotPink1","orchid2"
};
static void DrawActivity(IlcActivity activity,IlcDrawingBoard
                         drawingBoard,IlcInt id)
{
   drawingBoard.eraseGraphic(id);
   drawingBoard.drawArrow(id,
                      activity.getStartVariable().getMin(),
                      activity.getStartVariable().getMax(),
                      activity.getEndVariable().getMin(),
                      activity.getEndVariable().getMax(),
                      1,
                      colors[id % 21]);
}
ILCDEMON3(MyUpdateActivityDemon,IlcActivity,activity,IlcDrawingBoard,
          drawingBoard,IlcInt,id)
{
   DrawActivity(activity,drawingBoard,id);
}

ILCGOAL3(MyGoalReDrawActivityOnBacktrack,IlcActivity,
         activity,IlcDrawingBoard,drawingBoard,IlcInt,id)
{
   DrawActivity(activity,drawingBoard,id);
   return 0;
}

static void ActivityOnDrawingBoard(IlcActivity activity, IlcDrawingBoard
                                   drawingBoard)
{
   IloInt id = drawingBoard.makeNewGraphicObject();
   activity.getStartVariable().whenDomain(MyUpdateActivityDemon
                              (activity.getManager(),activity,drawingBoard,id));
   activity.getEndVariable().whenDomain(MyUpdateActivityDemon
                              (activity.getManager(),activity,drawingBoard,id));
   activity.getManager().addReversibleAction(MyGoalReDrawActivityOnBacktrack
                              (activity.getManager(),activity,drawingBoard,id));
}

ILCGOAL2(MyIlcDrawingBoardCreation, IlcDrawingBoard, drawingBoard,
         IlcScheduler, scheduler) {
   drawingBoard.clean();
   IloInt id = drawingBoard.makeNewGraphicObject();
   drawingBoard.drawLine(id,0,0,200,200,1);
   id = drawingBoard.makeNewGraphicObject();
   drawingBoard.drawArrow(id,0,0,0,200,1);
   id = drawingBoard.makeNewGraphicObject();
   drawingBoard.drawArrow(id,0,0,200,0,1);
   id = drawingBoard.makeNewGraphicObject();
   drawingBoard.drawLabel(id,170,50,"Min");
   id = drawingBoard.makeNewGraphicObject();
   drawingBoard.drawLabel(id,5,170,"Max");
```

```
   IlcActivityIterator it(scheduler);
   while (it.ok()) {
      IlcActivity activity = (*it);
      ActivityOnDrawingBoard(activity,drawingBoard);
      ++it;
   }
   return 0;
}

ILOCPGOALWRAPPER2(MyDrawingBoardCreation, solver, IlcDrawingBoard,
                  drawingBoard, IlcScheduler, scheduler){
   return MyIlcDrawingBoardCreation(solver, drawingBoard, scheduler);
}
```

The solving is then done as follows:

```
IlcScheduler scheduler(debugger.getSolver());
IlcDrawingBoard drawingBoard(debugger);
goal = IloAndGoal(solver.getEnv(),
                  MyDrawingBoardCreation(solver.getEnv(),
                                         drawingBoard,scheduler),
                  goal);
  while(debugger.initialize()) {
     solver.startNewSearch(goal);

...
```

Improving your Application

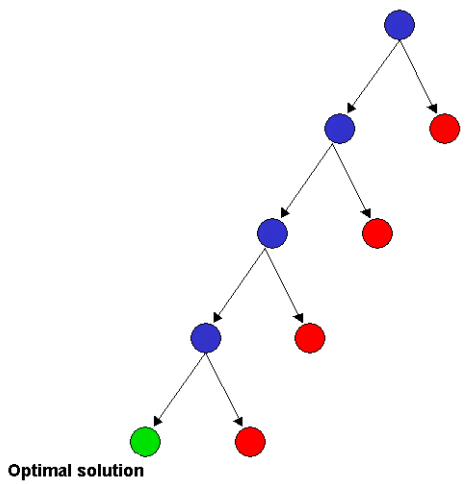
This section provides you with an example on how to improve an optimization application. This application contains an objective function, and uses ILOG Solver with the Depth First Search exploration order.

If your own application corresponds to these criteria, you can use the following visualization methodology to improve the search and the propagation. In this section, you will learn how to analyze right subtrees in your Search Tree and follow a tutorial to see how the propagation can be improved.

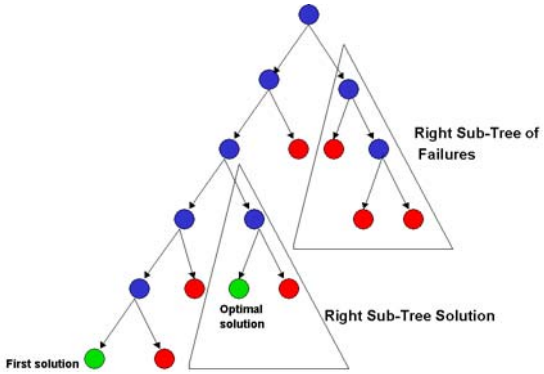
- Analyzing Right Subtrees
- Tutorial: Tuning the Propagation in the debuggolomb Example

Analyzing Right Subtrees

To interpret the results of the Search Tree, keep in mind the theoretical ideal shape of a Search Tree, which looks like the figure below.



To spot possible problems in your application, analyze the right subtrees in your Search Tree.



For a binary tree, considering that the right branch is the contradiction of the left branch, when right subtrees are developed you can draw one of the following two conclusions:

- Solutions appear in the right subtree, in which case the search can be improved since it is the contradiction of a decision that leads to a better solution.
- Solutions do not appear in the right subtree, in which case there is a lack of propagation because the back propagation of the cost should have pruned the subtree automatically.

When you use the Depth First Search exploration order, the higher in the tree the right subtrees are, the more you should pay attention to them, because the tree is potentially exponential in its depth.

Tutorial: Tuning the Propagation in the debuggolomb Example

This section is based on the debuggolomb example. It shows you how Solver Debugger can be used to tune the propagation of an ILOG Solver application.

Filter levels

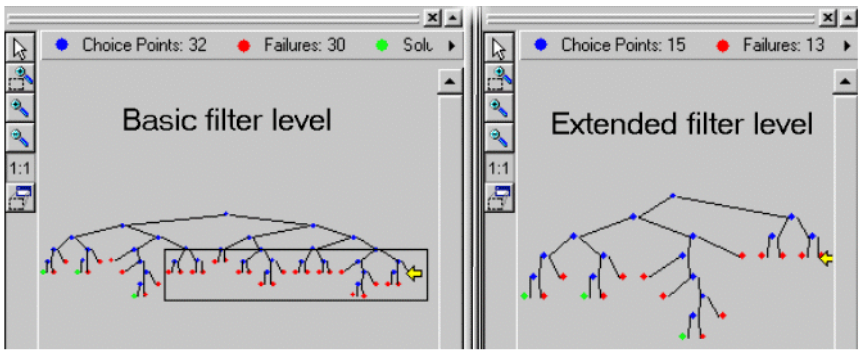
In order to tune the propagation in the debuggolomb example, we will use the `alldiff` global constraint of ILOG Solver with two different levels of propagation.

> ***Reminder:*** *When connecting to the debuggolomb example, pass two integer parameters: the size of the problem and the filtering level of the* `alldiff` *constraint.*
>
> `debuggolomb 6 0` *(0 = basic filtering level)*
>
> `debuggolomb 6 1` *(1 = medium filtering level)*
>
> `debuggolomb 6 2` *(2 = extended filtering level)*

- When using the basic filter level of this constraint, ILOG Solver guarantees that, at any computation point, the specified variables do not have the values of the already-assigned variables inside their domain.
- When using the extended filter level, ILOG Solver reasons on the domains instead of the values and guarantees that, for each value in the domain of any given variable, there exist values in the domains of the remaining variables such that the constraint is satisfied.

Using the Search Tree

Visualize the Search Trees corresponding to the basic and extended filter levels. The following figure represents the Search Trees corresponding to the two filter levels.



In the Search Tree, the extended level enforces a stronger pruning than the basic one, while the basic filter level produces a bigger tree with additional right subtrees.

These right subtrees have only failure leaves, which is a sign of lack of propagation. By setting the filter level to the extended level these right subtrees are pruned.

Using the Christmas Tree

The Christmas Tree provides a picture of the cost and the efficiency of the extended propagation.

Compare the number of propagation events in the first right subtree located in the frame in Figure 1.6 with the number of propagation events occurring at the corresponding big failure node in Figure 1.7.

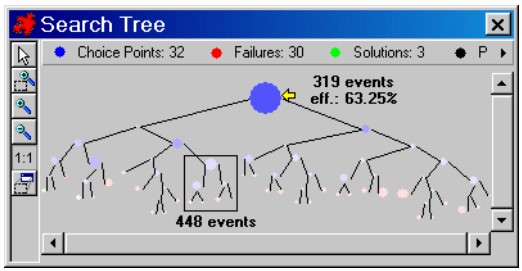


Figure 1.6 *The Christmas Tree for the debuggolomb problem with a basic level of propagation*

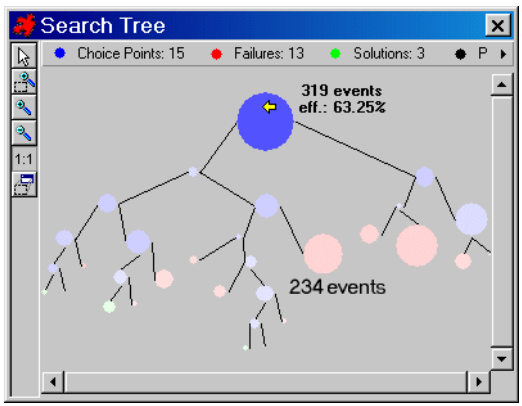


Figure 1.7 *The Christmas Tree for the debuggolomb problem with an extended level of propagation*

The big node requires two times fewer propagation events to detect the failure than the subtree. So the extended propagation saved time.

Now compare the Initial Propagation statistics by looking at the root node. The efficiency of domain reduction is the same (63.25%).

Using the Propagation Spy

Inspect the Initial Propagation. ILOG Solver adds a hidden constraint when posting the `alldiff` constraint. When tracing the propagation at the first big failure node of the extended filter level, the Propagation Spy displays the extra propagation, as shown in the figure below.

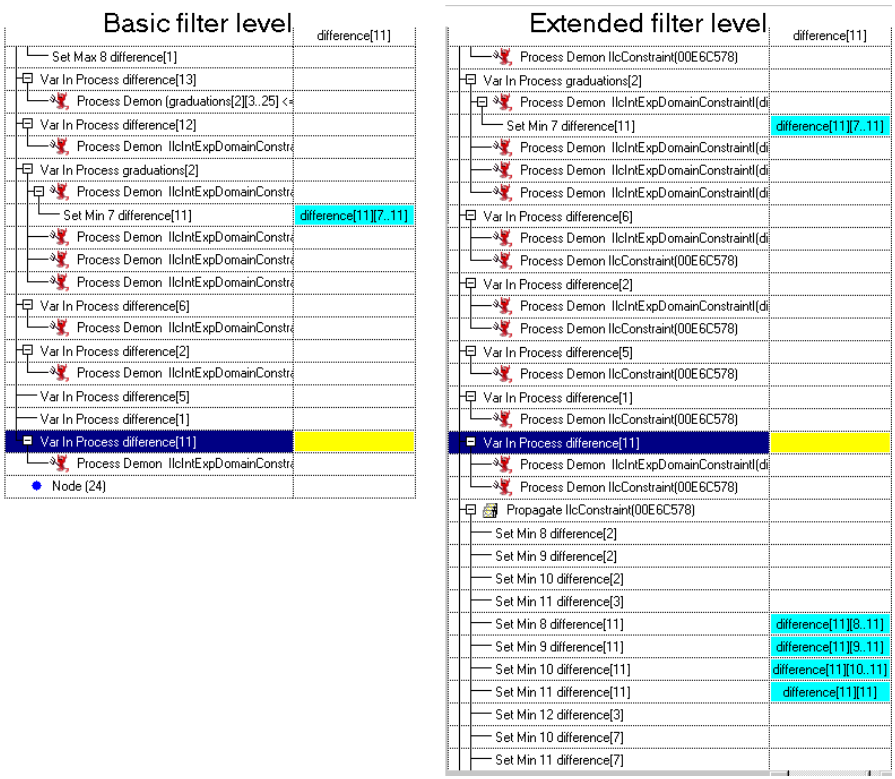


Figure 1.6 *Comparing two propagation traces using the Propagation Spy for the debuggolomb problem*

- With the basic filter level, the propagation stops after reducing the variable `difference[11]` to the interval `[7..11]`.
- With the extended level, the propagation of an additional internal constraint posted by the `alldiff` constraint strongly reduces the domains by means of `Set Min` events. This additional propagation leads to a failure, avoiding a subtree exploration.

Notice that four `Set Min` events are triggered on the `difference[11]` variable where one should be enough. These `Set Min` events are `Remove Value` orders that have been translated by ILOG Solver into `Set Min` because the value to remove was a bound.

So, reasoning on bounds instead of on domains is sufficient in this case. Tune the `alldiff` constraint to the intermediate filter level, which reasons on bounds instead of on domains.

You obtain the same Search Tree as with the extended filter level. The Propagation Spy shows that, at the same choice point, the variable `difference[11]` is bound more quickly.

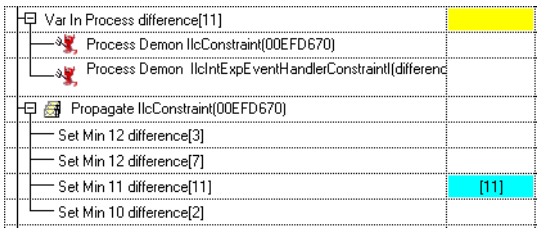
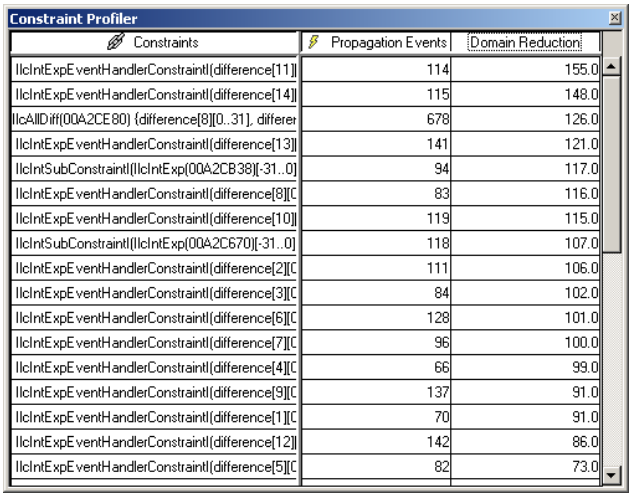


Figure 1.9 *Propagation Spy for the debuggolomb problem with an intermediate filter level*

Using the Constraint Profiler



You can paste the matrix from the Constraint Profiler to Microsoft Excel and make your own statistics. For instance, summing the propagation events gives the results shown below. The table contains the statistics on the number of events for the debuggolomb problem.

Filter Level	Total Number of Events
Basic	3534
Intermediate	2726
Extended	2790

Solver Debugger API

This section contains the C++ API for the ILOG Solver Debugger.

IlcAddOnDebugger

Category C++ class

Description The class `IlcAddOnDebugger` is the base class for the Debugger classes specialized in debugging an application based on an add-on for ILOG Solver.

Include File `<ilsolver/addondebugger.h>`

Synopsis

```
class IlcAddOnDebugger {
public:
    IlcAddOnDebugger(IlcAddOnDebuggerI* impl=0);
    IloBool displayResult() const;
    IlcAddOnDebuggerI* getImpl() const;
  };
#endif
```

Constructor

```
IlcAddOnDebugger(IlcAddOnDebuggerI* impl=0);
```

Creates a new handle class on the same implementation object.

Member Functions

```
IloBool displayResult() const;
```

Displays the result graphically. In the current implementation, by default, this member function is automatically called at each solution. In Scheduler applications it displays a Gantt chart with precedences.

```
IlcAddOnDebuggerI* getImpl() const;
```

Returns the implementation object.

IlcDebugger

Category C++ class

Description The class `IlcDebugger` is a handle class on an instance of the implementation class `IlcDebuggerI`. This class manages the debugging protocol with the Debugger GUI so that an external application or a specific engine can connect to the GUI.

Include File `<ilsolver/debugger.h>`

Synopsis

```
class IlcDebugger {
public:
     IlcDebugger(IlcDebuggerI* impl = 0);
     int beginNestedSearchTreeView();
     void breakExecution();
     void browseAll();
     void close();
     IloBool displayResult() const;
     int endNestedSearchTreeView();
     IlcDebuggerI* getImpl() const {return _impl;}
     IloBool isActive() const;
     IloBool isAutomaticBrowse() const;
     IloBool isAutomaticDisplay() const;
     IloBool isConnected() const;
     IloBool isTextual() const;
     int newSearchTreeView();
     void sendConsole(const char* text);
     void sendSolution(const char* text);
     void setAutomaticBrowse(IloBool automaticBrowse);
     void setAutomaticDisplay(IloBool automatic);
 };
```

Constructor **`IlcDebugger`**`(IlcDebuggerI* impl = 0);`

Creates a new handle class on the same implementation object.

Member Functions

`int` **`beginNestedSearchTreeView`**`();`

Creates a new Search Tree view in Solver Debugger. Sends notification of the next node created to the new Search Tree. Goes back to the previous Search Tree whenever `endNestedSearch` is called.

`void` **`breakExecution`**`();`

Interrupts the execution.

`void` **`browseAll`**`();`

Forces the Debugger to browse the model.

```
void close();
```

Closes the connection between the user application and the GUI server and releases the memory.

```
IloBool displayResult() const;
```

Displays the result graphically. In the current implementation, by default, this member function is automatically called at each solution. In Scheduler applications it displays a Gantt chart with precedences.

```
int endNestedSearchTreeView();
```

Goes back to the parent Search Tree. Considers the current Search Tree as terminated. Returns the identification of the parent Search Tree.

```
IlcDebuggerI* getImpl() const;
```

Returns the underlying implementation object.

```
IloBool isActive() const;
```

Returns true if the Debugger was set to active upon creation.

```
IloBool isAutomaticBrowse() const;
```

Returns true if the Debugger is in automatic browse mode.

```
IloBool isAutomaticDisplay() const;
```

Returns true if the Debugger is set to the automatic display mode for the result.

```
IloBool isConnected() const;
```

Returns true if the connection with the GUI server is still active.

```
IloBool isTextual() const;
```

Returns true if the Debugger was set to textual upon creation.

```
int newSearchTreeView();
```

Creates a new Search Tree view in Solver Debugger. Sends notification of the next node created to the new Search Tree. Considers the previous Search Tree as terminated. Returns the identification of the Search Tree view to be created.

```
void sendConsole(const char* text);
```

Sends a message to the Debugger Console Notebook Page.

```
void sendSolution(const char* text);
```

Sends a message to the Debugger Notebook Page.

```
void setAutomaticBrowse(IloBool automaticBrowse);
```

Tells the Debugger to switch on or off the automatic model browsing.

```
void setAutomaticDisplay(IloBool automatic);
```

Tells the Debugger to switch on or off the automatic display mode for the result. If the Debugger is set to the automatic display mode, add-on debuggers display a specific graphic representation of the solution, such as a Gantt chart for the Scheduler Debugger.

IlcDrawingBoard

Category C++ class

Description The class `IlcDrawingBoard` provides a 2D representation of the search algorithm.

Include File `<ilsolver/drawingboard.h>`

Synopsis

```
class IlcDrawingBoard
{
public:
  IlcDrawingBoard(IlcSolverDebugger debugger,
                  IloInt width = 100,
                  IloInt height = 100,
                  IloNum scaleH = 1.0,
                  IloNum scaleV = 1.0);
  IlcDrawingBoard(IlcDrawingBoardI* impl = 0);
  void clean();
  IlcIntExp constantInt(IloInt value) const;
  IlcFloatExp constantNum(IloNum value) const;
  void drawArc(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
               IloInt rectWidth, IloInt rectHeight,
               IlcFloat startAngle, IlcFloat angleRange,
               IloInt th);
  void drawArrow(IloInt id, IloInt x1, IloInt y1,
                 IloInt x2, IloInt y2, IloInt th,
                 const char* col = "black");
  void drawEllipse(IloInt id, IloInt xLowerLeft,
                   IloInt yLowerLeft, IloInt rectWidth,
                   IloInt rectHeight, IloInt th);
  void drawFilledArc(IloInt id, IloInt xLowerLeft,
                     IloInt yLowerLeft, IloInt rectWidth,
                     IloInt rectHeight, IlcFloat startAngle,
                     IlcFloat angleRange,
                     const char* col, IloInt th);
  void drawFilledEllipse(IloInt id, IloInt xLowerLeft,
                         IloInt yLowerLeft, IloInt rectWidth,
                         IloInt rectHeight, const char* col,
                         IloInt th);
  void drawFilledLabel(IloInt id, IloInt x, IloInt y,
                       const char* str, const char* col);
  void drawFilledPolygon(IloInt id, IloInt x[], IloInt y[],
                         IloInt count, const char* col,
                         IloInt th, IloInt fillRule);
  void drawFilledRectangle(IloInt id, IloInt xLowerLeft,
                           IloInt yLowerLeft,
                           IloInt rectWidth,
                           IloInt rectHeight,
                           const char* col, IloInt th);
```

```
void drawFilledRoundedRectangle(IloInt id,  IloInt xLowerLeft,
                                IloInt yLowerLeft,
                                IloInt rectWidth,
                                IloInt rectHeight,
                                IlcFloat rc,
                                const char* col,
                                IloInt th);
void drawGrid(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
              IloInt rectWidth, IloInt rectHeight,
              IloInt cols, IloInt rows, IloInt th);
void drawLabel(IloInt id, IloInt x, IloInt y, const char* str);
void drawLine(IloInt id, IloInt x1, IloInt y1,
              IloInt x2, IloInt y2, IloInt th,
              const char* col = "black");
void drawPolyline(IloInt id, IloInt x[], IloInt y[],
                  IloInt count, IloInt th);
void drawRectangle(IloInt id, IloInt xLowerLeft,
                   IloInt yLowerLeft, IloInt rectWidth,
                   IloInt rectHeight, IloInt th);
void drawRoundedRectangle(IloInt id, IloInt xLowerLeft,
                          IloInt yLowerLeft,
                          IloInt rectWidth,
                          IloInt rectHeight,
                          IlcFloat rc, IloInt th);
void eraseGraphic(IloInt id);
void makeArc(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
             IlcIntExp rectWidth, IlcIntExp rectHeight,
             IlcFloatExp startAngle,
             IlcFloatExp angleRange,IlcIntExp th);
void makeArrow(IlcIntExp x1, IlcIntExp y1,
               IlcIntExp x2, IlcIntExp y2,
               IlcIntExp th, const char* col = "black");
void makeEllipse(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                 IlcIntExp rectWidth, IlcIntExp rectHeight,
                 IlcIntExp th);
void makeFilledArc(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                   IlcIntExp rectWidth, IlcIntExp rectHeight,
                   IlcFloatExp startAngle,
                   IlcFloatExp angleRange,
                   const char* col, IlcIntExp th);
void makeFilledEllipse(IlcIntExp xLowerLeft,
                       IlcIntExp yLowerLeft,
                       IlcIntExp rectWidth,
                       IlcIntExp rectHeight,
                       const char* col, IlcIntExp th);
void makeFilledLabel(IlcIntExp x, IlcIntExp y,
                     const char* str, const char* col);
void makeFilledPolygon(IlcIntVarArray x,IlcIntVarArray y,
                       const char* col, IlcIntExp th,
                       IlcIntExp fillRule);
```

```
  void makeFilledRectangle(IlcIntExp xLowerLeft,
                           IlcIntExp yLowerLeft,
                           IlcIntExp rectWidth,
                           IlcIntExp rectHeight,
                           const char* col, IlcIntExp th);
  void makeFilledRoundedRectangle(IlcIntExp xLowerLeft,
                                  IlcIntExp yLowerLeft,
                                  IlcIntExp rectWidth,
                                  IlcntExp rectHeight,
                                  IlcFloatExp rc,
                                  const char* col,
                                  IlcIntExp th);
  void makeGrid(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                IlcIntExp rectWidth, IlcIntExp rectHeight,
                IlcIntExp cols, IlcIntExp rows,
                IlcIntExp th);
  void makeLabel(IlcIntExp x, IlcIntExp y,const char* str);
  void makeLine(IlcIntExp x1, IlcIntExp y1,
                IlcIntExp x2, IlcIntExp y2,
                IlcIntExp th,const char* col = "black");
  int  makeNewGraphicObject();
  void makePolyline(IlcIntVarArray x, IlcIntVarArray y,
                    IlcIntExp th);
  void makeRectangle(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                     IlcIntExp rectWidth, IlcIntExp rectHeight,
                     IlcIntExp th);
  void makeRoundedRectangle(IlcIntExp xLowerLeft,
                            IlcIntExp yLowerLeft,
                            IlcIntExp rectWidth,
                            IlcIntExp rectHeight,
                            IlcFloatExp rc, IlcIntExp th);
  void setBackgroundColor(const char* col);
};
```

Constructors

```
IlcDrawingBoard(IlcSolverDebugger debugger,
                IloInt width = 100,
                IloInt height = 100,
                IloNum scaleH = 1.0,
                IloNum scaleV = 1.0);
```

Creates a new Drawing Board. The `width` parameter indicates the width of the panel when undocked. The `height` parameter indicates the height of the panel when undocked. The parameter `scaleH` has a value between 0 and 2. It indicates the horizontal scale factor. The parameter `scaleV` has a value between 0 and 2. It indicates the vertical scale factor.

```
IlcDrawingBoard(IlcDrawingBoardI* impl = 0);
```

Creates a new handle class on the same Drawing Board implementation object.

Member Functions

```
void clean();
```

Deletes all information from the Drawing Board.

```
IlcIntExp constantInt(IloInt value) const;
```

Creates an `IlcIntExp` object containing an integer constant.

```
IlcFloatExp constantNum(IloNum value) const;
```

Creates an `IlcFloatExp` object containing a floating-point constant.

```
void drawArc(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
             IloInt rectWidth, IloInt rectHeight,
             IlcFloat startAngle, IlcFloat angleRange,
             IloInt th);
```

Creates a nonbacktrackable arc identified by`id` and embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The angle of the arc begins at position `startAngle` and moves, in a clockwise direction, the number of degrees indicated by `angleRange`. The line of the arc is of thickness `th`.

```
void drawArrow(IloInt id, IloInt x1, IloInt y1, IloInt x2, IloInt y2,
               IloInt th, const char* col="black");
```

Creates a nonbacktrackable arrow identified by `id`. The arrow starts at `(x1,y1)` and ends at `(x2,y2)`. The thickness of the line is `th` and the default color of the arrow is black.

```
void drawEllipse(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
                 IloInt rectWidth, IloInt rectHeight, IloInt th);
```

Creates a nonbacktrackable empty ellipse identified by `id` and embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The outline of the ellipse is of thickness `th`.

```
void drawFilledArc(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
                   IloInt rectWidth, IloInt rectHeight,
                   IlcFloat startAngle, IlcFloat angleRange,
                   const char* col, IloInt th);
```

Creates a nonbacktrackable filled arc identified by`id` and embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The angle of the arc begins at position `startAngle` and moves, in a clockwise direction, the number of degrees indicated by `angleRange`. The line of the arc is of thickness `th` and the segment is filled with the color `col`.

```
void drawFilledEllipse(IloInt id, IloInt xLowerLeft,
                       IloInt yLowerLeft, IloInt rectWidth,
                       loInt rectHeight, const char* col,
                       IloInt th);
```

Creates a nonbacktrackable filled ellipse identified by`id` and embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The outline of the ellipse is of thickness `th` and the ellipse is filled with the color `col`

```
void drawFilledLabel(IloInt id, IloInt x, IloInt y,
                     const char* str, const char* col);
```

Creates a nonbacktrackable filled label identified by`id`. The label is anchored at the point (`x`,`y`), contains the character string `str` and is of the color specified by `col`.

```
void drawFilledPolygon(IloInt id, IloInt x[], IloInt y[],
                       IloInt count, const char* col,
                       IloInt th, IloInt fillRule);
```

Creates a nonbacktrackable polygon identified by`id` and filled with the color `col`. The parameter `th` indicates the thickness of the line. The two integer array parameters `x[]`, `y[]` specifying the X and Y coordinates must be of the same size. The `count` parameter indicates the number of point coordinates. The `fillRule` parameter indicates which points are to be considered as being inside the polygon, depending on the number of crossing segments that define the shape of the area to be filled. There are two rules to define the shape:

- setting the `fillRule` parameter to 0 takes the “EvenOdd Rule”
- setting the `fillRule` parameter to 1 takes the “Winding Rule”.

EvenOdd Rule: according to this rule, in the case of the complex polygon shown below, the central area of the star is not considered to lie inside the polygon and, therefore, is not filled. This is the default value.

Winding Rule: according to this rule, the central area of the star is considered to lie inside the polygon and, therefore, is filled.

```
void drawFilledRectangle(IloInt id, IloInt xLowerLeft,
                         IloInt yLowerLeft, IloInt rectWidth,
                         IloInt rectHeight, const char* col,
                         IloInt th);
```

Creates a nonbacktrackable rectangle identified by `id` and filled with the color `col`. The rectangle is anchored at the point (`xLowerLeft`, `yLowerLeft`) and its dimensions are `rectWidth`, `rectHeight`. The outline of the rectangle is of thickness `th`.

```
void drawFilledRoundedRectangle(IloInt id, IloInt xLowerLeft,
                                IloInt yLowerLeft,
                                IloInt rectWidth,
                                IloInt rectHeight,
                                IlcFloat rc, const char* col,
                                IloInt th);
```

Creates a nonbacktrackable rectangle with rounded corners and filled with the color `col`. The rectangle is identified by `id`. The parameters are similar to those of `drawFilledRectangle` above, except for `rc`, which specifies the amplitude of the arc used for the rounded corners.

```
void drawGrid(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
              IloInt rectWidth, IloInt rectHeight,
              IloInt cols, IloInt rows, IloInt th);
```

Creates a nonbacktrackable grid identified by `id` and embedded in a rectangle. The rectangle is specified by the anchor points `xLowerLeft`, `yLowerLeft`, and the dimensions `rectWidth`, `rectHeight`. The columns and rows in the grid are specified by the integers `cols` and `rows`. The lines of the grid are of thickness `th`.

```
void drawLabel(IloInt id, IloInt x, IloInt y, const char* str);
```

Creates a nonbacktrackable label identified by `id`. The label is anchored at the point (`x`,`y`) and contains the character string `str`.

```
void drawLine(IloInt id, IloInt x1, IloInt y1, IloInt x2, IloInt y2,
              IloInt th, const char* col = "black");
```

Creates a nonbacktrackable line identified by `id`. The line, drawn between the two points (`x1`,`y1`) and (`x2`,`y2`), is of thickness `th` and default color black.

```
void drawPolyline(IloInt id, IlcInt x[], IlcInt y[],
                  IlcInt count, IlcInt th);
```

Creates a nonbacktrackable polyline identified by `id`. The two integer array parameters `x[]`, `y[]` specifying the X and Y coordinates must be of the same size. The `count` parameter indicates the number of point coordinates. The `th` parameter indicates the thickness of the line. To obtain an empty polygon, give the same value to the first X and last X and to the first Y and last Y.

```
void drawRectangle(IloInt id, IloInt xLowerLeft, IloInt yLowerLeft,
                   IloInt rectWidth, IloInt rectHeight, IloInt th);
```

Creates a nonbacktrackable empty rectangle identified by `id`. The rectangle is anchored at the point (`xLowerLeft`, `yLowerLeft`) and its dimensions are `rectWidth`, `rectHeight`. The outline of the rectangle is of thickness `th`.

```
void drawRoundedRectangle(IloInt id, IloInt xLowerLeft,
                          IloInt yLowerLeft, IloInt rectWidth,
                          IloInt rectHeight, IlcFloat rc,
                          IloInt th);
```

Creates a nonbacktrackable empty rectangle with rounded corners, identified by `id`. The parameters are similar to those of `drawRectangle` above, except for `rc`, which specifies the amplitude of the arc used for the rounded corners.

```
void eraseGraphic(IloInt id);
```

Erases a nonbacktrackable graphic identified by `id`.

```
void makeArc(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
             IlcIntExp rectWidth, IlcIntExp rectHeight,
             IlcFloatExp startAngle, IlcFloatExp angleRange,
             IlcIntExp th);
```

Creates a backtrackable arc embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The angle of the arc begins at position `startAngle` and moves, in a clockwise direction, the number of degrees indicated by `angleRange`. The line of the arc is of thickness `th`.

```
void makeArrow(IlcIntExp x1, IlcIntExp y1, IlcIntExp x2, IlcIntExp y2,
               IlcIntExp th, const char* col = "black");
```

Creates a backtrackable arrow that starts at the point (`x1,y1`) and ends at the point (`x2,y2`). The thickness of the line is `th` and the default color of the arrow is black.

```
void makeEllipse(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                 IlcIntExp rectWidth, IlcIntExp rectHeight,
                 IlcIntExp th);
```

Creates a backtrackable empty ellipse embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The outline of the ellipse is of thickness `th`.

```
void makeFilledArc(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                   IlcIntExp rectWidth, IlcIntExp rectHeight,
                   IlcFloatExp startAngle, IlcFloatExp angleRange,
                   const char* col, IlcIntExp th);
```

Creates a backtrackable filled arc embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The angle of the arc begins at position `startAngle` and moves, in a clockwise direction, the number of degrees indicated by `angleRange`. The line of the arc is of thickness `th` and the segment is filled with the color `col`.

```
void makeFilledEllipse(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                       IlcIntExp rectWidth, IlcIntExp rectHeight,
                       const char* col, IlcIntExp th);
```

Creates a backtrackable filled ellipse embedded in a rectangle. The rectangle is specified by the lower left coordinates `xLowerLeft`, `yLowerLeft`, the width `rectWidth` and the height `rectHeight`. The outline of the ellipse is of thickness `th` and the ellipse is filled with the color `col`

```
void makeFilledLabel(IlcIntExp x, IlcIntExp y,
                     const char* str, const char* col);
```

Creates a backtrackable filled label. The label is anchored at the point (`x`, `y`), contains the character string `str` and has the background color specified by `col`.

```
void makeFilledPolygon(IlcIntVarArray x, IlcIntVarArray y,
                       const char* col, IlcIntExp th,
                       IlcIntExp fillRule);
```

Creates a backtrackable polygon filled with the color `col`. The `th` parameter indicates the line thickness. The two integer array parameters `x`, `y` specifying the X and Y coordinates, must be of the same size. The `fillRule` parameter indicates which points are to be considered as being inside the polygon, depending on the number of crossing segments that define the shape of the area to be filled. There are two rules to define the shape:

- setting the `fillRule` parameter to 0 takes the “EvenOdd Rule”
- setting the `fillRule` parameter to 1 takes the “Winding Rule”.

See the method `drawFilledPolygon` for an explanation of these rules.

```
void makeFilledRectangle(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                         IlcIntExp rectWidth, IlcIntExp rectHeight,
                         const char* col, IlcIntExp th);
```

Creates a backtrackable rectangle filled with the color `col`. The rectangle is anchored at the point (`xLowerLeft`, `yLowerLeft`) and its dimensions are `rectWidth`, `rectHeight`. The outline of the rectangle is `th` pixels thick.

```
void makeFilledRoundedRectangle(IlcIntExp xLowerLeft,
                                IlcIntExp yLowerLeft,
                                IlcIntExp rectWidth,
                                IlcIntExp rectHeight,
                                IlcFloatExp rc,
                                const char* col, IlcIntExp th);
```

Creates a backtrackable rectangle with rounded corners. The parameters are similar to those of `makeFilledRectangle` above, except for `rc`, which specifies the amplitude of the arc used for the rounded corners.

```
void makeGrid(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
              IlcIntExp rectWidth, IlcIntExp rectHeight,
             IlcIntExp cols, IlcIntExp rows, IlcIntExp th);
```

Creates a backtrackable grid embedded in a rectangle. The rectangle is specified by the anchor points `xLowerLeft`, `yLowerLeft`, and the dimensions `rectWidth`, `rectHeight`. The columns and rows in the grid are specified by the integers `cols` and `rows`. The lines of the grid are of thickness `th`.

```
void makeLabel(IlcIntExp x, IlcIntExp y, const char* str);
```

Creates a backtrackable label anchored at the point `(x,y)` and containing the character string `str`.

```
void makeLine(IlcIntExp x1, IlcIntExp y1, IlcIntExp x2, IlcIntExp y2,
              IlcIntExp th, const char* col = "black");
```

Creates a backtrackable line between the two points `(x1,y1)` and `(x2,y2)`, of thickness `th` and default color black.

```
int  makeNewGraphicObject();
```

Creates a new nonbacktrackable 2D graphic object.

```
void makePolyline(IlcIntVarArray x, IlcIntVarArray y, IlcIntExp th);
```

Creates a backtrackable polyline.The two integer array parameters `x`, `y`, specifying the X and Y coordinates, must be of the same size. The `th` parameter indicates the thickness of the line. To obtain an empty polygon, give the same value to the first X and last X and to the first Y and last Y.

```
void makeRectangle(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                   IlcIntExp rectWidth, IlcIntExp rectHeight,
                   IlcIntExp  th);
```

Creates a backtrackable empty rectangle. The rectangle is anchored at the point `(xLowerLeft, yLowerLeft)` and its dimensions are `rectWidth`, `rectHeight`. The outline of the rectangle is `th` pixels thick.

```
void makeRoundedRectangle(IlcIntExp xLowerLeft, IlcIntExp yLowerLeft,
                          IlcIntExp rectWidth, IlcIntExp rectHeight,
                          IlcFloatExp rc, IlcIntExp th);
```

Creates a backtrackable empty rectangle with rounded corners. The parameters are similar to those of `makeRectangle` above, except for `rc`, which specifies the amplitude of the arc used for the rounded corners.

```
void setBackgroundColor(const char* col);
```

The parameter `col` sets the background color for the Drawing Board.

IlcSchedulerDebugger

Category C++ class

Description The class `IlcSchedulerDebugger` provides an interface to the ILOG Solver Debugger in an ILOG Scheduler environment.

Include File `<ilsched/schedulerdebugger.h>`

Synopsis

```
class IlcSchedulerDebugger: public IlcAddOnDebugger {
public:
  IlcSchedulerDebugger(IlcSolverDebugger debugger,
                       IlcSchedule s,
                       IloBool traceAllActivities = IloTrue,
                       IloBool traceAllResources  = IloTrue,
                       IloBool traceAllowed = IloTrue);
  IlcSchedulerTrace getSchedulerTrace() const;
  void registerActivity(IloActivity activity,
                        IloBool toBrowse = IloTrue,
                        IloBool toTrace = IloTrue,
                        IlcVisualizeOption displayDomain =
                                           IlcVisualizeDisabled);
  void registerResource(IloResource resource,
                        IloBool toBrowse = IloTrue);
  void registerUnaryResourceArray(IloArray<IloUnaryResource>
                                           &resources,
                                  const char* name,
                                  IloBool toBrowse = IloTrue);
  void setHorizon(IlcInt horizon);
 };
```

Constructor

```
IlcSchedulerDebugger(IlcSolverDebugger debugger,
                       IlcSchedule s,
                       IloBool traceAllActivities = IloTrue,
                       IloBool traceAllResources  = IloTrue,
                       IloBool traceAllowed = IloTrue);
```

The parameter `s` sets the `IlcSchedule` you have created to declare your activities and resources. If the parameter `traceAllActivities` is true and `traceAllowed` is also true, all activities are traced. If the parameter `traceAllResources` is true and `traceAllowed` is also true, all resources are traced. If the parameter `traceAllowed` is true, a specific trace for a Scheduler application is added.

Member Functions

```
IlcSchedulerTrace getSchedulerTrace() const;
```

Returns the graphical Scheduler trace added by the instance `IlcSchedulerDebugger`.

```
void registerActivity(IloActivity activity,
       IloBool toBrowse = IloTrue,
       IloBool toTrace = IloTrue,
        IlcVisualizeOption displayDomain = IlcVisualizeDisabled);
```

Registers an interest in an activity. The parameter `activity` sets the `IlcActivity` to consider. If the parameter `toBrowse` is true, the model will display the activity in the Debugger Model Browser. If the parameter `toTrace` is true, the activity must be traced and the `start` and `end` variables must be taken into account in the Christmas Tree. If the parameter `displayDomain` is true, the activity will be monitored graphically in the Activity Domains panel.

```
void registerResource(IloResource resource,
                      IloBool toBrowse = IloTrue);
```

Registers an interest in a resource. The parameter `resource` indicates the `IloResource` to consider. If the parameter `toBrowse` is true, the option Browse Active Model will be activated in the Debugger Model Browser.

```
void registerUnaryResourceArray(IloArray<IloUnaryResource>&resources,
                           const char* name,
                           IloBool toBrowse = IloTrue);
```

Registers an interest in a unary resource array. The parameter `resources` indicates the array of unary resources to consider. The parameter `name` indicates the name of the array to be displayed in the browser. If the parameter `toBrowse` is true, the option Browse Active Model will be activated in the Debugger Model Browser.

```
void setHorizon(IlcInt horizon);
```

Gives the horizon to consider in the graphical representation of the Activity Domains. This member function is to be called after registering the activities. The parameter `horizon` sets the maximum value of the upper bounds of all activity end variables.

IlcSolverDebugger

Category C++ class

Description The class `IlcSolverDebugger` provides an interface to the ILOG Solver Debugger in an ILOG Solver environment.

Include File `<ilsolver/solverdebugger.h>`

Synopsis

```
class class IlcSolverDebugger: public IlcDebugger {
public:
    IlcSolverDebugger(IloSolver solver,
                      IloBool active = IloTrue,
                      IloBool textual = IloFalse,
                      double pollingPeriod = 1.0,
                      int socketPort = 4011,
                      const char* guiServer = 0,
                      IloBool nestedSearch = IloFalse);
    void breakIfBound(IloFloatVar var, IloNum val);
    void breakIfBound(IloIntVar var, IloInt val);
    void breakIfDomainMaxLe(IloFloatVar var, IloNum val);
    void breakIfDomainMaxLe(IloIntVar var, IloInt val);
    void breakIfDomainMinGe(IloFloatVar var, IloNum val);
    void breakIfDomainMinGe(IloIntVar var, IloInt val);
    void displayProfiling();
    IloSolver getSolver() const;
    IloBool initialize();
    void profileAllConstraints(IloBool all);
    void registerConstraint(IloConstraint cst,
                            IloBool browse = IloTrue,
                            IloBool trace = IloTrue);
    void registerVariable(IloFloatVar var,
              IloBool browse = IloTrue,
              IloBool trace = IloTrue,
               IlcVisualizeOption option = IlcVisualizeDisabled);
    void registerVariable(IloFloatVarArray var,
               IloBool browse = IloTrue,
               IloBool trace = IloTrue,
                IlcVisualizeOption option = IlcVisualizeDisabled,
                const char* name = 0);
    void registerVariable(IloIntVar var,
              IloBool browse = IloTrue,
              IloBool trace = IloTrue,
               IlcVisualizeOption option = IlcVisualizeDisabled);
    void registerVariable(IloIntVarArray var,
               IloBool browse = IloTrue,
               IloBool trace = IloTrue,
               IlcVisualizeOption option = IlcVisualizeDisabled,
               const char* name = 0);
  };
```

Constructor

```
IlcSolverDebugger(IloSolver solver,
                  IloBool active = IloTrue,
                  IloBool textual = IloFalse,
                  double pollingPeriod = 1.0,
                  int socketPort = 4011,
                  const char* guiServer = 0,
                  IloBool nestedSearch = IloFalse);
```

When the `active` parameter is true, the Debugger is active and tries to connect to the GUI immediately. Otherwise, the Debugger library has no effect. In this case, the member function `initialize()` returns true when it is called for the first time and false otherwise. The `sendConsole()` and `sendSolution()` member functions display the messages on the standard output. The parameter `solver` indicates the `IloSolver` object to which the constraints are added. With the parameter `textual`, if the Debugger is active, XML messages are sent to the standard output (`stdout`) instead of the GUI. The parameter `pollingPeriod` sets the number of seconds between two abort or break checks. The parameter `socketPort` indicates the port number for which the GUI is waiting. The parameter `guiServer` indicates the machine name on which the GUI is running. If this parameter is null, then `localhost` is taken.

Member Functions

```
void breakIfBound(IloFloatVar var, IloNum val);
```

Registers a conditional breakpoint. The execution is interrupted at this conditional breakpoint if the specified floating point variable is bound to `val`.

```
void breakIfBound(IloIntVar var, IloInt val);
```

Registers a conditional breakpoint. The execution is interrupted at this conditional breakpoint if the specified integer variable is bound to `val`.

```
void breakIfDomainMaxLe(IloFloatVar var, IloNum val);
```

Registers a conditional breakpoint. The execution is interrupted at this conditional breakpoint if the domain `max` of the specified floating point variable is lower than or equal to `val`.

```
void breakIfDomainMaxLe(IloIntVar var, IloInt val);
```

Registers a conditional breakpoint. The execution is interrupted at this conditional breakpoint if the domain `max` of the specified integer variable is lower than or equal to `val`.

```
void breakIfDomainMinGe(IloFloatVar var, IloNum val);
```

Registers a conditional breakpoint. The execution is interrupted at this conditional breakpoint if the domain `min` of the specified floating point variable is greater than or equal to `val`.

```
void breakIfDomainMinGe(IloIntVar var, IloInt val);
```

Registers a conditional breakpoint. The execution is interrupted at this conditional breakpoint if the domain `min` of the specified integer variable is greater than or equal to `val`.

```
void displayProfiling();
```

Creates and displays the Constraint Profiler window from code. (Equivalent to clicking on the Constraint Profiler button from the GUI).

```
IloSolver getSolver() const;
```

Returns the same `IloSolver` instance as the one passed to the constructor.

```
IloBool initialize();
```

Initializes a running session. Returns false if a `Quit` order comes from the GUI.

```
void profileAllConstraints(IloBool all);
```

By default all constraints are profiled. To restrict the list of monitored constraints, use `profileAllConstraints(IloFalse)` and call the `registerConstraint` method on the constraints your are interested in.

```
void registerConstraint(IloConstraint cst,
                        IloBool browse = IloTrue,
                        IloBool trace = IloTrue);
```

Registers an interest in the constraint `cst`. The parameter `cst` indicates the constraint to consider. If the parameter `browse` is true, the model should be displayed in the Debugger Model Browser. If the parameter `trace` is true, the variables must be traced.

```
void registerVariable(IloFloatVar var,
                      IloBool browse = IloTrue,
                      IloBool trace = IloTrue,
                        IlcVisualizeOption option = IlcVisualizeDisabled);
```

Registers an interest in the float variable `var`. The parameter `var` indicates the variable to consider. If the parameter `browse` is true, the model should be displayed in the Debugger Model Browser. If the parameter `trace` is true, the variables must be traced. The parameter `option` gives access to a graphical monitoring of the variable (displaying its range or value).

```
void registerVariable(IloFloatVarArray var,
                      IloBool browse = IloTrue,
                      IloBool trace =IloTrue,
                      IlcVisualizeOption option = IlcVisualizeDisabled,
                      const char* name = 0);
```

Registers an interest in the array of float variables `var`. The parameter `var` indicates the array of variables to consider. If the parameter `browse` is true, the model should be displayed in the Debugger Model Browser. If the parameter `trace` is true, the variables must be traced. The parameter `option` gives access to a graphical monitoring of the variable (displaying its range or value).

```
void registerVariable(IloIntVar var,
                      IloBool trace =IloTrue,
                      IloBool browse = IloTrue,
                        IlcVisualizeOption option = IlcVisualizeDisabled);
```

Registers an interest in the integer variable `var`. The parameter `var` indicates the `Ilovar` to consider. If the parameter `browse` is true, then if the model is browsed it will be displayed in the Debugger Model Browser. If the parameter `trace` is true, variables must be traced and taken into account in the Christmas Tree. The parameter `option` gives access to a graphical monitoring of the variable (displaying its range or value).

```
void registerVariable(IloIntVarArray var,
                       IloBool browse = IloTrue,
                       IloBool trace = IloTrue,
                         IlcVisualizeOption option = IlcVisualizeDisabled,
                       const char* name = 0);
```

Registers an interest in the array of integer variables `var`. The parameter `var` indicates the array of variables to consider. If the parameter `browse` is true, the model should be displayed in the Debugger Model Browser. If the parameter `trace` is true, the variables must be traced and taken into account in the Christmas Tree. The parameter `option` gives access to a graphical monitoring of the variable (displaying its domain, range, and value).

IlcVisualizeOption

Category Type Definition

Include File `<ilsolver/debuggerbasics.h>`

Description The type definition `IlcVisualizeOption` is a visualization option that can be passed to `IlcSolverDebugger.registerVariable` or `IlcSchedulerDebugger.registerActivity`.

Synopsis

```
typedef unsigned int IlcVisualizeOption;
typedef enum  {IlcVisualizeDisabled = 0,
               IlcVisualizeDomain = 1,
               IlcVisualizeInterval = 2,
               IlcVisualizeValue = 4,
               IlcVisualizeBoolean = 8} IlcVisualizeOptionType;
```

`IlcVisualizeOption` can take the above values.

Index

A

Abort button **20**
Activity Domains **36**
adding breakpoints in your code **33**
application
  connecting **18**
  connecting remotely **19**
  connecting two at the same time **19**
  customizing **14**, **15**
  improving **47**

B

backtracking **41**
`beginNestedSearchTreeView` member function
  `IlcDebugger` Debugger C++ class **59**
boolean variable, displaying **34**
Break button **20**
`breakExecution` member function
  `IlcDebugger` Debugger C++ class **59**
`breakIfBound(IloFloatVar...)` member function
  `IlcSolverDebugger` Debugger C++ class **76**
`breakIfBound(IloIntVar...)` member function
  `IlcSolverDebugger` Debugger C++ class **76**
`breakIfDomainMaxLe(IloFloatVar...)`
  `IlcSolverDebugger` Debugger C++ class **76**
`breakIfDomainMaxLe(IloIntVar...)`
  `IlcSolverDebugger` Debugger C++ class **76**
`breakIfDomainMinGe(IloFloatVar...)`
  `IlcSolverDebugger` Debugger C++ class **76**
`breakIfDomainMinGe(IloIntVar...)`
  `IlcSolverDebugger` Debugger C++ class **77**
breakpoints **23**
buttons
  Abort **20**
  Break **20**
  Continue **20**
  Continue Trace Till Next Node **20**
  Decision Stack **21**
  Inspect Current Node **21**
  Propagation Spy **21**
  Run **18**, **20**
  Step Into **20**
  Step Out **20**
  Step Over **20**
  View Constraint Profiler **21**
  View Search Tree **21**

C

choice point **21**, **52**
Christmas Tree
  creating **38**
  domain reduction **40**
  interpreting **40**
  number of propagation events **40**
  visualizing **38**

`clean` member function
`IlcDrawingBoard` C++ class **66**
`close` member function
`IlcDebugger` Debugger C++ class **60**
colors
Drawing Board **43**
event types **31**
node types **23**
Propagation Spy **31**
Variable Domains **35**
commands to control the Debugger **20**
conditional breakpoints **33**
connecting
an application **18**
an application remotely **19**
two applications **19**
`constantInt` member function
`IlcDrawingBoard` C++ class **66**
`constantNum` member function
`IlcDrawingBoard` C++ class **66**
Constraint Profiler
description **40**
using **52**
constraint statistics **41**
Continue button **20**
Continue Trace Till Next Node button **20**
customizing your application
ILOG Scheduler C++ application **15**
ILOG Solver C++ application **14**

D

Debugger
commands **20**
examples directory **12**, **13**
Tree Inspector **38**
debuggolomb example
description **13**
filter levels **18**, **49**
Search Tree **22**
debugjobshopm example **14**, **44**
debuglsknight example **13**
debugnqueen example
Decision Stack **27**
description **13**
Drawing Board **42**
Variable Domains **34**
debugpheasant example
description **14**
Initial Propagation **31**
Decision Stack button **21**
Decision Stack docked panel **26**
decisions taken at a node **27**
`displayResult` member function
`IlcAddOnDebugger` Debugger C++ class **57**
`IlcDebugger` Debugger C++ class **60**
distribution directories **12**
domain reduction **41**
`drawArc` member function
`IlcDrawingBoard` C++ class **66**
`drawArrow` member function
`IlcDrawingBoard` C++ class **66**
`drawEllipse` member function
`IlcDrawingBoard` C++ class **66**
`drawFilledArc` member function
`IlcDrawingBoard` C++ class **66**
`drawFilledEllipse` member function
`IlcDrawingBoard` C++ class **67**
`drawFilledLabel` member function
`IlcDrawingBoard` C++ class **67**
`drawFilledPolygon` member function
`IlcDrawingBoard` C++ class **67**
`drawFilledRectangle` member function
`IlcDrawingBoard` C++ class **68**
`drawFilledRoundedRectangle` member function
`IlcDrawingBoard` C++ class **68**
`drawGrid` member function
`IlcDrawingBoard` C++ class **68**
Drawing Board
advanced use **43**
basic use **42**
customizing colors **43**
example **44**
Drawing Board class **63**
`drawLabel` member function
`IlcDrawingBoard` C++ class **68**
`drawLine` member function
`IlcDrawingBoard` C++ class **68**
`drawPolyline` member function
`IlcDrawingBoard` C++ class **68**

drawRectangle member function
IlcDrawingBoard C++ class **69**
drawRoundedRectangle member function
IlcDrawingBoard C++ class **69**

E

endNestedSearchTreeView member function
IlcDebugger Debugger C++ class **60**
eraseGraphic member function
IlcDrawingBoard C++ class **69**
event types, colors **31**
examples
debuggolomb **13**, **18**, **22**, **49**
debugjobshopm **14**, **44**
debuglsknight **13**
debugnqueen **13**, **27**, **34**, **42**
debugpheasant **14**, **31**
description **13**
examples directory **12**, **13**

G

Gantt chart **14**, **28**
getImpl member function
IlcAddOnDebugger Debugger C++ class **57**
IlcDebugger Debugger C++ class **60**
getSchedulerTrace member function
IlcSchedulerDebugger Debugger C++ class **74**
getSolver member function
IlcSolverDebugger Debugger C++ class **77**
GUI
idle mode **18**
initializing mode **17**
GUI installation **12**

I

idle mode, Debugger GUI **18**
IlcAddOnDebugger Debugger C++ class
constructor **57**, **59**
displayResult **57**
getImpl **57**
IlcDebugger Debugger C++ class **59**
beginNestedSearchTreeView **59**
breakExecution **59**
close **60**
constructor **59**
displayResult **60**
endNestedSearchTreeView **60**
getImpl **60**
isActive **60**
isAutomaticDisplay **60**
isConnected **60**
isTextual **60**
newSearchTreeView **60**
sendConsole **60**
sendSolution **60**
setAutomaticDisplay **61**
IlcDrawingBoard C++ class **63**
clean **66**
constantInt **66**
constantNum **66**
constructor **65**
drawArc **66**
drawArrow **66**
drawEllipse **66**
drawFilledArc **66**
drawFilledEllipse **67**
drawFilledLabel **67**
drawFilledPolygon **67**
drawFilledRectangle **68**
drawFilledRoundedRectangle **68**
drawGrid **68**
drawLabel **68**
drawLine **68**
drawPolyline **68**
drawRectangle **69**
drawRoundedRectangle **69**
eraseGraphic **69**
makeArc **69**
makeArrow **69**
makeEllipse **69**
makeFilledArc **70**
makeFilledEllipse **70**
makeFilledLabel **70**
makeFilledPolygon **70**
makeFilledRectangle **70**
makeFilledRoundedRectangle **71**
makeGrid **71**

makeLabel **71**
makeLine **71**
makeNewGraphicObject **71**
makePolyline **71**
makeRectangle **71**
makeRoundedRectangle **72**
setBackgroundColor **72**
IlcSchedulerDebugger Debugger C++ class
constructor **73**
getSchedulerTrace **74**
registerActivity **74**
registerResource **74**
registerUnaryResourceArray **74**
setHorizon **74**
IlcSolverDebugger Debugger C++ class
breakIfBound(IloFloatVar..) **76**
breakIfBound(IloIntVar...) **76**
breakIfDomainMaxLe(IloFloatVar...) **76**
breakIfDomainMaxLe(IloIntVar...) **76**
breakIfDomainMinGe(IloFloatVar...) **76**
breakIfDomainMinGe(IloIntVar...) **77**
constructor **76**
getSolver **77**
initialize **77**
registerConstraint **77**
registerVariable(IloBool...) **78**
registerVariable(IloFloatVar...) **77**
registerVariable(IloFloatVarArray...) **78**
registerVariable(IloIntVarArray...) **78**
IlcVisualizeOption type definition **79**
improving your application **47**
include files
for Debugger with Scheduler **13**
for Debugger with Solver **12**
ILOG Scheduler C++ application **15**
ILOG Solver C++ application **14**
initial propagation **40**
Initial Propagation, tracing **32**
initialize member function
IlcSolverDebugger Debugger C++ class **77**
initializing mode, Debugger GUI **17**
Inspect Current Node button **21**
installation
Debugger GUI **12**
online documentation **12**
Scheduler examples **13**
Solver examples **12**
installation directories **12**
integer variable
bounds **34**, **35**
domain **34**
state **35**
value **34**, **35**
interpreting
Christmas Tree **40**
right subtrees **47**
isActive member function
IlcDebugger Debugger C++ class **60**
isAutomaticDisplay member function
IlcDebugger Debugger C++ class **60**
isConnected member function
IlcDebugger Debugger C++ class **60**
isTextual member function
IlcDebugger Debugger C++ class **60**

L

local search **28**

M

makeArc member function
IlcDrawingBoard C++ class **69**
makeArrow member function
IlcDrawingBoard C++ class **69**
makeEllipse member function
IlcDrawingBoard C++ class **69**
makeFilledArc member function
IlcDrawingBoard C++ class **70**
makeFilledEllipse member function
IlcDrawingBoard C++ class **70**
makeFilledLabel member function
IlcDrawingBoard C++ class **70**
makeFilledPolygon member function
IlcDrawingBoard C++ class **70**
makeFilledRectangle member function
IlcDrawingBoard C++ class **70**
makeFilledRoundedRectangle member function
IlcDrawingBoard C++ class **71**

`makeGrid` member function
`IlcDrawingBoard` C++ class **71**
`makeLabel` member function
`IlcDrawingBoard` C++ class **71**
`makeLine` member function
`IlcDrawingBoard` C++ class **71**
`makeNewGraphicObject` member function
`IlcDrawingBoard` C++ class **71**
`makePolyline` member function
`IlcDrawingBoard` C++ class **71**
`makeRectangle` member function
`IlcDrawingBoard` C++ class **71**
`makeRoundedRectangle` member function
`IlcDrawingBoard` C++ class **72**
mode selectors **24**
monitoring
local search **28**
nested search **29**
restart **28**
solution restoring **28**
monitoring decisions **25**
Current Node Inspector docked panel **26**
Decision Stack docked panel **26**
Node Inspector dialog box **26**
Propagation Spy **26**

N

nested search **29**
`newSearchTreeView` member function
`IlcDebugger` Debugger C++ class **60**
Node Inspector dialog box **23**, **26**
node life cycle **23**
node types
colors **23**
root node **23**
status **23**

O

online documentation **12**

P

propagation
events **30**, **41**
statistics **38**
tracing at a specific node **32**
visualizing **30**
Propagation Spy
button **21**
colors **31**
description **26**
event types **31**
using **30**, **51**

R

`registerActivity` member function
`IlcSchedulerDebugger` Debugger C++ class **74**
`registerConstraint` member function
`IlcSolverDebugger` Debugger C++ class **77**
`registerResource` member function
`IlcSchedulerDebugger` Debugger C++ class **74**
`registerUnaryResourceArray` member function
`IlcSchedulerDebugger` Debugger C++ class **74**
`registerVariable(IloFloatVar...)` m.f.
`IlcSolverDebugger` Debugger C++ class **77**
`registerVariable(IloFloatVarArray...)` m.f.
`IlcSolverDebugger` Debugger C++ class **78**
`registerVariable(IloIntVar...)` m.f.
`IlcSolverDebugger` Debugger C++ class **78**
`registerVariable(IloIntVarArray...)` m.f.
`IlcSolverDebugger` Debugger C++ class **78**
remote connection **19**
restart **28**
right subtrees, interpreting **47**
root node **23**
Run button **18**, **20**

S

Search Tree
description **22**
ideal shape **47**
mode selectors **24**
node inspector **23**, **26**

`sendConsole` member function
`IlcDebugger` Debugger C++ class **60**
`sendSolution` member function
`IlcDebugger` Debugger C++ class **60**
`setAutomaticDisplay` member function
`IlcDebugger` Debugger C++ class **61**
`setBackgroundColor` member function
`IlcDrawingBoard` C++ class **72**
`setHorizon` member function
`IlcSchedulerDebugger` Debugger C++ class **74**
solution restoring **28**
Solver Debugger
C++ libraries **11**
starting the GUI **17**
starting ILOG Solver Debugger **17**
statistics
for a constraint **41**
propagation **38**
Step Into button **20**
Step Out button **20**
Step Over button **20**

T

tool bar
execution **20**
standard **20**
tracing
Initial Propagation **32**
propagation at a specific node **32**
propagation events **30**
Tree Inspector **38**
type definition `IlcVisualizeOption` **79**

V

Variable Domains
colors **35**
disabling visualization **34**
displaying Boolean variables **34**
integer variable bounds **34**, **35**
integer variable domain **34**
integer variable value **34**, **35**
setting visualization options **33**
state of integer variable interpreted as Boolean **35**
visualizing **34**
View Constraint Profiler button **21**
View Search Tree button **21**
visualizing
Activity Domains **36**
both propagation events and search **38**
propagation **30**
search **21**
Variable Domains **34**